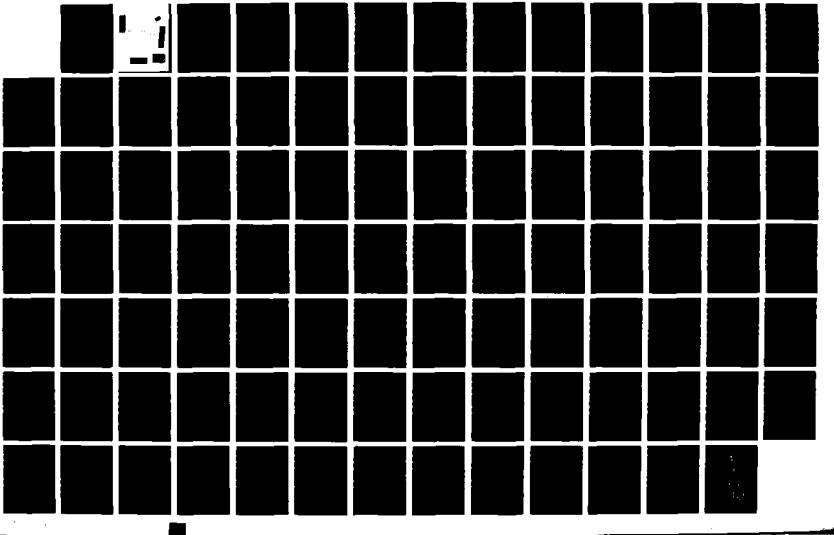
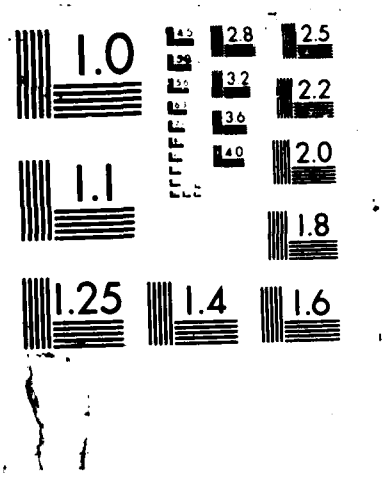
AD-A187 090 RECOVERY USING VIRTUAL MEMORY(U) MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE E K KOLODNER JUL 87 MIT/LCS/TR-404 N00014-83-K-0125 1/1

UNCLASSIFIED F/G 12/6 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

MIT/LCS/TR-404

RECOVERY USING VIRTUAL MEMORY

Elliot K. Kolodner

July 1987

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

AD-A187098

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-404	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Recovery Using Virtual Memory

12. PERSONAL AUTHOR(S)
Kolodner, Elliot K.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1987 July	91

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
atomic actions, recovery, garbage collection, virtual memory, persistent storage, stable storage, distributed systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Maintaining the consistency of long-lived on-line data in the presence of failures is important for many applications such as airline reservation and banking systems. After a crash, the long-lived data must be recovered for the application to continue running. Storing the data and later restoring it is the job of a recovery system. This thesis presents a new recovery method with two features: it is fast because as much as possible it uses data already stored by an application in virtual memory for recovery, and it is novel because it allows data in virtual memory to be organized in a heap with automatic garbage collection. The recovery method is designed to be used in the Argus system, but it will also work for other persistent storage systems.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

U.S. Government Printing Office: 1985—607-047

Recovery Using Virtual Memory

by

Elliot K. Kolodner

June 1987

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125, and in part by the National Science Foundation under grant DCR-8503662.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Recovery Using Virtual Memory

by

Elliot K. Kolodner

Abstract

Maintaining the consistency of long-lived on-line data in the presence of failures is important for many applications such as airline reservation and banking systems. After a crash, the long-lived data must be recovered for the application to continue running. Storing the data and later restoring it is the job of a recovery system. This thesis presents a new recovery method with two features: it is fast because as much as possible it uses data already stored by an application in virtual memory for recovery, and it is novel because it allows data in virtual memory to be organized in a heap with automatic garbage collection. The recovery method is designed to be used in the Argus system, but it will also work for other persistent storage systems.

This report is a minor revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on May 8, 1987 in partial fulfillment of the requirements for the Degree of Master of Science.

Thesis Supervisor: Barbara H. Liskov

Title: Nippon Electric Company Professor of Software Science and Engineering

Keywords: atomic actions, recovery, garbage collection, virtual memory, persistent storage, stable storage, distributed systems

Acknowledgments

I would like to thank my thesis advisor, Barbara Liskov, for listening to my ideas and helping me to organize and edit this thesis.

I would also like to thank all of the members of the Programming Methodology Group, past and present, who discussed various aspects of the research with me: Dorothy Curtis, Mark Day, Debbie Hwang, Paul Johnson, Rivka Ladin, Gary Leavens, Brian Oki, Sharon Perl, Bob Scheifler, and William Weihl.

Finally, I thank my family and my wife, Elana, for providing support and encouragement.

Contents

List of Figures

Chapter 1

Introduction

Maintaining the consistency of long-lived on-line data in the presence of failures is important for many applications such as airline reservation and banking systems. For example, account information for a banking system must not be lost even if the computer storing it crashes. After a crash, the account information must be recovered for the banking system to continue running. Storing this information and later restoring it is the job of a recovery system. This thesis presents a new recovery method with two features: it is fast because as much as possible it uses data already stored by a program in virtual memory for recovery, and it allows data in virtual memory to be organized in a heap with automatic garbage collection. The recovery method is designed to be used in the Argus system[16,17], but it will also work for other systems that organize resilient data in a garbage-collected heap.

Argus programs run as *atomic actions*. Actions mask failures that may occur while they are running. An action either completes entirely and *commits*, or it is guaranteed to have no effect and *aborts*. A crash or other failure before an action commits forces it to abort. Therefore, the recovery system only needs to save and restore the effects of committed actions.

When an Argus program runs, its data is stored in virtual memory. Virtual memory contains volatile storage and it alone is not sufficient to support recovery. A way of ensuring that data survives crashes is to write it to a *stable storage* device. A stable storage device avoids the loss of information despite failure with very high probability. Stable storage can be organized as a log of completed actions with writing to one end. To recover, the log is

read backwards until all data has been restored to virtual memory. A previous recovery system for Argus[21,22] does all recovery from the log assuming that all of virtual memory is lost in a crash. The problem with this approach is that recovery is slow. The amount of processing is proportional to the number of committed actions, which grows without bound. Recovery time can be shortened by periodically rewritting the log to remove information about old actions whose effects have already been overwritten by newer ones. However, this rewritting is an expensive process and can only be done rarely. Therefore, the log is typically quite long.

This thesis investigates an approach to recovery based on the assumption that most of virtual memory survives crashes. In particular, the approach distinguishes between *hard* and *soft* crashes. Virtual memory uses main memory as a cache for the slower backing store. Usually main memory is semi-conductor and is volatile, whereas the backing store is on disk and is non-volatile. A hard crash is a crash in which both components of virtual memory are corrupted. The old recovery method must be used for a hard crash. A soft crash is a crash in which the volatile main memory is lost or corrupted, but the non-volatile backing store survives uncorrupted. After a soft crash, the new recovery method reconstructs the data in virtual memory by using the surviving backing store and reading just enough of the log to restore the part of the virtual memory that was not written to disk recently. The amount of log that needs to be read can be kept small by periodic *checkpoints*. A checkpoint is taken by flushing all dirty pages of the main memory to the backing store. (A page of main memory is dirty if it has not been written to the backing store since it was last modified.) Checkpoints can be taken frequently because they are relatively inexpensive.

Virtual memory is organized in Argus as a heap of objects using dynamic memory allocation and garbage collection. Such an organization is desirable because it eases the job of programming by freeing the programmer from concerns about storage management. However, it poses several problems for recovery. First, objects move when the heap is compacted. Enough information must be available after a crash to find objects on the surviving backing store. Second, a crash might occur during garbage collection. A garbage collection algorithm is presented that allows the resilient data to be recovered from the backing store even if a crash occurs during garbage collection.

This thesis proposes a method for recovery using both virtual memory and a log after a soft crash. The method is faster than using a log alone. The scheme is novel because it allows virtual memory to be organized as a heap.

1.1 Related Work

A major point of difference between Argus and other transaction systems is the approach taken to storage management. Other systems require explicit system calls to create, modify or delete resilient data (data that needs to survive crashes). They do not use garbage collection. Argus is unique among transaction systems in that it integrates resilient data into the fabric of a programming language and allows objects to become or cease being resilient implicitly. An Argus programmer divides data between stable state and volatile state; only the stable state is resilient. The stable state consists of all objects accessible from a stable root. Inter-object references can change dynamically under program control, and objects enter and leave the stable state implicitly. Furthermore programmers never deallocate storage explicitly; storage is reclaimed and compacted using garbage collection. Thus, a recovery method designed for Argus is responsible for determining which objects are stable and need to be written to stable storage. If the method recovers using virtual memory, it must also be able to find objects in virtual memory after a crash and recover from a crash occurring in the middle of garbage collection.

1.1.1 The Current Argus Recovery System

The recovery system closest to the one described in this thesis is the current Argus recovery system designed by Oki[21,22]. Both systems organize stable storage as a log. The organization of the log, the method for detecting what objects need to be written to the log, the method for writing to the log and the method for processing the log after a crash are similar. However, the current recovery system assumes that all crashes are hard crashes. After every crash, it discards virtual memory and reads the entire log on stable storage to restore the resilient data to a fresh copy of virtual memory. A goal in the design of the current recovery system was that it be implementable for the Argus prototype[18]. Recovery using virtual memory requires changes to the operating system that the implementers of the

prototype wanted to avoid. A more extensive comparison between the two systems appears in chapter 6.

1.1.2 Other Recovery Systems

Like the recovery scheme presented in this thesis, recovery methods for other transaction systems also differentiate between hard and soft crashes, and provide fast recovery for soft crashes. These systems usually use one of two main methods for recovery: shadows or write-ahead logging. First the recovery method proposed in this thesis for Argus is compared with recovery in System R[9], a relational database system developed at IBM that uses shadows. Then it is compared with recovery in TABS[7,23,24], a distributed transaction system developed at CMU that uses write-ahead logging.

System R

System R maintains its data in a file system and not in virtual memory. A hard crash is called a media failure and refers to a corruption of the file system. A soft crash is called system restart and refers to a crash in which the file system survives uncorrupted.

System R uses a combination of a shadowing mechanism and a log to implement recovery. A shadow page and a current page are maintained for each active file page. Modifications are made to the current page and recorded in the log so that they can be undone or redone for transaction abort and recovery. Because the unit of locking (record) is not the same as the unit of recovery (file page) in System R, each modification to a file has to be recorded in the log to allow concurrent transactions to update the same file. At regular intervals checkpoints are taken to shorten the portion of the log that must be processed after a crash and reclaim the disk storage used by shadows for active pages. At a checkpoint pages in volatile store are flushed to their current pages and each current page replaces its paired shadow page.

After implementing the system, the designers of System R argued that they should have used a write-ahead log instead of shadows. A write-ahead log would have allowed the system to update file pages in place and reduced storage requirements. Using shadows did not save writing to the log; even using shadows, every modification to a file page had to be recorded

in the log. Using shadows also required an extra level of indirection in the file system–a map pointing to the pages of the file. The indirection meant greater access time for both random and sequential access. The indirection also made taking a checkpoint expensive because the maps had to be updated and storage had to be freed at every checkpoint.

The recovery method presented in this thesis uses a shadowing mechanism without incurring the extra writing to the log and the overhead for checkpoints. This is because it uses shadows for logical entities (objects) and not physical entities (pages). Thus, it does not have to record every modification to an object in the log. Instead it only needs to record an object version in the log when an action that modified the object commits. It also does not have to wait until a checkpoint to replace shadow versions by current versions. It does the replacement at action commit for each object modified by the committing action. Thus, checkpoints are a lot cheaper; they only involve flushing the dirty pages of the volatile store (main memory) to the non-volatile store (backing store).

TABS

TABS uses virtual memory to recover. The data server is the basic unit of computation in TABS. A data server encapsulates data and provides operations on that data. A data server's recoverable data resides in a disk file called a recoverable segment that is mapped into the server's virtual address space. A soft crash in TABS is called a node failure. A hard crash is called a media failure.

TABS uses a write-ahead log to implement recovery. The TABS server library provides routines to manipulate objects in the recoverable segment. To modify an object the server calls routines to obtain a write lock on the object, write the object to the log and pin the object in main memory. The write-ahead log protocol requires that the object be pinned in main memory (i.e., not written to the backing store) until the version is physically in the log. Before the transaction modifying the object commits, a routine has to be called to write the modified version to the log. The write-ahead log protocol requires that the modified versions of all objects updated by a transaction be physically in the log before the transaction commits.

Shadows are more appropriate for an Argus recovery system than a write-ahead log.

First, maintaining object versions in virtual memory simplifies the implementation of nested actions and allows actions to be aborted quickly.

Second, using a write-ahead log to allow update in place would require more writing to the log than what is required for shadows. As mentioned earlier, an Argus heap contains both volatile and stable objects. Using a write-ahead log, modifications to all objects whether volatile or stable would have to be recorded in the log to allow recovery from transaction abort. With shadows, an object is recorded in the log only when it is stable and an action that has modified it commits. Also, recording an object in the log involves more than just writing a physical copy of the object to the log. References to contained objects have to be changed from virtual memory references to references that still have meaning after garbage collection or a hard crash.

Lastly, some types of Argus objects are dynamic and can grow (or shrink) in size. Update in place does not work well for objects that grow in size. The indirection provided by the shadowing mechanism is appropriate for objects that grow dynamically.

1.2 Overview

Chapter 2 presents an overview of Argus highlighting those aspects that affect the recovery system. Chapter 3 presents an algorithm for garbage collection that allows recovery using virtual memory even if a crash occurs in the middle of garbage collection. Chapter 4 presents a method for recovery using virtual memory. Chapter 5 presents several optimizations to the recovery scheme, an alternative for recovery that uses a different method to find objects in virtual memory after a crash, and an alternative for atomic garbage collection. Finally chapter 6 concludes with a summary of the thesis, an evaluation of the scheme for recovery, and a discussion of future work.

Chapter 2

Argus

This chapter provides details about Argus necessary to understand the design of the recovery system. The first section discusses the programming language and system. The next section discusses action processing. Finally the last section discusses several assumptions about the Argus implementation and restrictions on the language that are required by the recovery method.

2.1 Argus Programming Language and System

The Argus[16,17] language and system allows a programmer to write programs that will execute on a distributed network of computers. The nodes of the network are independent computers each consisting of one or more processors with local memory and input output devices. Nodes may communicate with each other only by sending messages over the network.

The *guardian* is the basic module of an Argus program; a guardian is an abstraction of a processor and its memory. Each guardian resides at a single node, although a single node can support several guardians. A guardian encapsulates (or guards) long-lived resilient data and a set of processes that can operate on that data. Access to the guardian's data is granted only through operations, called *handlers*, which are defined in the body of the guardian definition. A guardian may also have background processes that operate on its data. An Argus program is a collection of guardians that communicate with each other through handler calls.

A guardian's state is divided into *stable* and *volatile* components. In a guardian definition

the programmer declares certain variables to be stable. A guardian's stable variables are collected into a single object called the *stable root*. The stable state is identified as all objects accessible from the stable root; those objects are called *accessible*. The stable root is created and initialized when the guardian is created and its stable variables are initialized. When a guardian crashes, its stable state survives, whereas its volatile state and processes are lost. When a guardian recovers from a crash, its stable state must be reconstructed before it resumes receiving handler calls and running its background processes.

Atomic actions are used to structure computation in an Argus program. Actions are *serializable* and *total*. Serializability means that when actions are executed concurrently, the effect will be as if they were run sequentially in some order. Totality means that an action is all or nothing, i.e., it completes entirely or is guaranteed to have no effect.

An action is initiated at one guardian and can spread to other guardians through handler calls. Totality requires that when an action completes, it either commits at every guardian that participated in the action or aborts at every participating guardian. If an action commits, all of its changes to the stable state are installed and become visible to other actions; if it aborts, all of its changes are discarded. Two-phase commit[9] (discussed below) is used to ensure totality.

Atomicity of actions is ensured through atomic objects. Actions are guaranteed to be atomic only if all of the data they share with other actions are atomic objects. Argus provides built-in atomic objects and user-defined atomic objects. There are two kinds of built-in atomic objects: immutable and mutable. Immutable built-in objects are atomic because their values never change. The values of mutable built-in atomic objects and user-defined atomic objects can change and the Argus system provides mechanisms to ensure that the changes are atomic.

For the mutable built-in atomic objects, read and write locks are used together with versions and a strict two-phase locking discipline to insure serializability and totality. To use a built-in atomic object, an action invokes one of the object's operations. The action acquires a lock on the object in the mode appropriate to the operation and holds the lock until it commits or aborts. When a write lock is first obtained for an action, a copy of the object is made in volatile memory and the action operates on this copy, which is called

the *current version*. The previous version, called the *base version*, is also retained. If the action commits, the current version becomes the base version, a copy of the current version is written to stable storage if the object is accessible, and the old base version is discarded. If the action aborts, the current version is discarded.

User-defined atomic objects allow greater concurrency than that achievable with built-in ones. They are constructed using *mutex* objects, which are containers for objects of arbitrary type. Mutex is a type generator that provides for mutual exclusion and recovery. A program uses the *seize* statement to obtain exclusive access to a mutex object. A program can cause an accessible mutex object to be written to stable storage by calling *changed*, an operation of the mutex type generator. Currently Argus guarantees two properties for the *changed* operation:

1. When *changed* is called from within an action, the object contained in the mutex will be written to stable storage after the call, but before the action commits.
2. If *changed* is called by an action for several mutexes at a single guardian, either all of the mutex versions written to stable storage on behalf of the action will be recovered after a crash or none of them will.

To insure the consistency of the mutex version written to stable storage, the recovery system seizes the mutex object to gain exclusive access to it while it is being written[17,26].

Actions in Argus are nested, so that there are both *topactions*, which are not nested, and *subactions*, which are. In particular, handler calls run as subactions of the calling action. Subactions require extensions to the locking and version management rules given above. However, these extensions are not significant to the recovery system. The recovery system carries out two-phase commit and writes to stable storage only for the commit of a topaction. When a topaction commits, only two versions exist for each modified object: the base version, and the current version that records all the changes made to the object by the action and its descendants. Therefore, nested actions are ignored in the remainder of this thesis.

2.2 Transaction Processing and Two-Phase Commit

As an action executes, it reads and modifies atomic objects at several guardians. Each guardian keeps track of the use made of its local objects. In particular, for every action that

visits a guardian, the guardian maintains a set of local objects that the action has modified. This set is called the *Modified Object Set* (MOS). A built-in atomic object is inserted in the MOS by the system when a write lock is obtained on it; a mutex is inserted when the program calls the *changed* operation on it. A separate MOS is maintained for each action that has visited the guardian and has not yet committed or aborted.

The recovery system needs a way of determining which objects in the MOS are accessible and need to be written to stable storage. It maintains, for each guardian, an *Accessibility Set* (AS) of resilient objects accessible from the guardian's stable variables. The AS is implemented by a single bit in each resilient object; the recovery system checks the bit to determine whether a resilient object in the MOS needs to be written to stable storage.

Maintaining the AS requires procedures for: initialization, determining when an object is *newly accessible* and must be inserted in the AS, and determining when an object is no longer accessible. The AS is initialized when a guardian is created by traversing the graph of objects accessible from the stable variables. Objects can become accessible only as a result of a modification to an object that was previously accessible. Thus, newly accessible objects are detected and inserted in the AS by examining the objects written to stable storage when an action commits (including the newly accessible objects themselves). Determining when an object is no longer accessible can only be accomplished by a full traversal of the stable state. This is done as part of garbage collection. Note that the AS is actually a superset of the objects that are accessible.

When an action completes, the system makes sure that it commits at every guardian it visited or that it aborts at every guardian. If the action commits, its changes to the stable state are made permanent by recording them on stable storage. The standard two-phase commit protocol[9] is used for this purpose. The guardian where an action originates is called the *coordinator*; the guardians visited through handler calls are the *participants*. A brief description of the protocol follows. There are two phases at the coordinator and two phases at participants. An action for which the first phase at a participant has been completed is called *prepared* at that participant. All messages sent during the protocol contain a system-wide unique *action identifier* identifying the action for which the protocol is being carried out.

At the Coordinator

Preparing Phase The coordinator sends a *prepare* message to each participant in the action (including itself). Then the coordinator waits for replies. If each participant replies with the *prepared* message, the coordinator enters the committing phase. If any participant replies with the *aborted* message, then the action is aborted. In that case, the coordinator sends the *abort* message to each participant. The coordinator can decide unilaterally at any time during the preparing phase to abort, e.g., if it does not receive responses from all participants even after retransmission has been tried.

Committing Phase If the coordinator has received a *prepared* message from each participant, it enters the committing phase. First the *committing* record containing the guardian identifiers of the participants is written to stable storage. This is the point at which the action has committed. Then the coordinator sends *committing* messages to each of the participants and waits for replies. When the *committed* message has been received from each of the participants, the coordinator writes the *done* record to stable storage and the protocol is complete.

At a Participant

Prepare Phase When a participant receives the *prepare* message from the coordinator, it begins its prepare phase. If the action is unknown at the participant (a crash might have occurred between the handler call and two-phase commit), it replies to the coordinator with the *aborted* message. Otherwise, the action prepares. The current versions of all stable objects modified by the action (all objects listed in the MOS that are also in the AS), the base versions of all objects made newly accessible by the action, and the current versions of all newly accessible objects that are in the MOS of another prepared action are written to stable storage. Then all read locks held by the action are released and a *prepared* record is written to stable storage. Then the participant sends the coordinator the *prepared* message and enters the completion phase.

Completion Phase In the completion phase, the participant waits for the coordinator to send a *commit* or *abort* message. If the participant receives the *commit* message, it writes the *committed* record to stable storage, releases write locks and replaces base versions with current versions for built-in atomic objects, and sends the committed message to the coordinator. If the participant receives the *abort* message, it writes the *aborted* record to stable storage, and releases write locks and discards current versions for built-in atomic objects. If the participant has not received a message from its coordinator, it can send a *query* message to the coordinator to find out the action's outcome.

2.3 Assumptions About Argus

Several assumptions are made in this thesis about the hardware and operating system of the machine for which the recovery system is being designed, and about the Argus implementation.

The design is for a standard architecture, a general purpose register machine with virtual memory (e.g., a VAX[1]). No special purpose hardware to support the recovery system or garbage collection is assumed.

The disks used for the backing store for virtual memory use error correcting or redundancy codes and can detect disk blocks that are bad or whose information was garbled since the last write. This means that hard crashes are detectable.

Soft crashes occur as a result of a software (for example, inconsistent data structures in the operating system) or hardware (for example, power failures) failure at a guardian's node. For a soft crash, it is assumed that a guardian crashes before bad information is written to its backing store.

There is no special hardware available for stable storage; stable storage is implemented by mirrored disks[14,13], and reading and writing stable storage is more expensive than ordinary reading or writing from a disk. It is important that writing to stable storage be fast to keep the cost of two-phase commit low. Sequential access to a disk is cheaper than random access. Because records are appended sequentially to a log, a log is a suitable organization for stable storage. If a different implementation of stable storage were available, other organizations for the information on stable storage could be considered.

The operating system provides support for Argus and recovery using virtual memory. In most operating systems the backing store on disk for virtual memory is considered volatile and does not survive crashes. To support recovery using virtual memory, the operating system will have to manage backing storage on disk similar to the way that storage for files is managed. The operating system will also have to provide primitives for virtual memory that allow control over paging. This assumption is discussed further in chapter 3.

Each guardian has its own virtual address space that is shared by the guardian's processes. This is the way Argus is currently implemented[18] and is a motivation for recovery

[1]VAX is a trademark of Digital Equipment Corporation

using virtual memory.

At every node, there is a special privileged guardian called the *guardian manager*. The guardian manager is responsible for creating new guardians at the node, keeping track of the resources used by guardians at the node, such as stable storage, and restarting guardians after a crash. For recovery using virtual memory, the guardian manager keeps track of disk storage used by guardians at the node for backing store and the mapping between virtual memory blocks and backing store disk blocks.

Two changes to the Argus language are assumed. The first change restricts the types of the stable variables. The type of a stable variable must be a type that can be guaranteed to be resilient. The resilience of an Argus type can be guaranteed if its representation is

1. a built-in atomic type,
2. an instance of a built-in atomic type generator instantiated by a resilient type, or
3. an instance of a mutex type generator instantiated by any type, or
4. a user-defined type whose representation is ultimately one of the above.

These types are called *resilient types*, and an object whose type is resilient is called a *resilient object*. Resilience for objects of these types can be guaranteed because modifications to the objects are coordinated with the recovery system. This restriction means that an object of a non-atomic mutable type (for example, an array) can become accessible only if it is enclosed in a mutex. Because Argus is a statically typed language, this restriction can be enforced by type checking at compile and configuration time. (*Configuration* is a step in the creation of executable code for a guardian between compilation and linking during which implementations are chosen for each of the modules making up a guardian.)

The second change simplifies the recovery semantics for mutex. The all or none property cited earlier for the recovery of mutexes is dropped from the language. This property required that if *changed* is called by an action for several mutexes at a single guardian, either all of the mutex versions written to stable storage on behalf of that action will be recovered after a crash or none of them will. Instead, the recovery system guarantees that all of the mutex versions will be recovered only if the action commits. If the action does not commit, some of the mutex versions may be recovered while others will not.

The designers of Argus included the all or none property to give programmers more expressive power when writing implementations for user defined atomic types. However, in practice it is difficult to make use of this property, and the property has not yet been used. By dropping the property from the language, recovery for mutexes is simpler and more efficient.

These changes are discussed further in chapter 6.

Chapter 3

Atomic Garbage Collection

Garbage collection is one of the main problems that needs to be to solved in order to make recovery using virtual memory possible.

Before the advent of virtual memory, the purpose of garbage collection was to reclaim storage in the heap no longer being used by a program. In virtual memory systems, garbage collection acquired the added purpose of improving paging performance. Paging performance is usually improved by compacting the storage being used in order to achieve a higher density of useful objects per page.

Compaction changes memory – objects are moved and pointers to objects are updated to reflect the moves. Most methods for reclaiming storage also modify objects in an effort to reduce the amount of additional storage needed to run the garbage collection algorithm itself. Thus, a crash in the middle of garbage collection would leave the backing store of virtual memory in a state that would make recovery using virtual memory impossible.

Recovery using virtual memory introduces a new requirement for garbage collection: the stable state cached in virtual memory must be recoverable even if a crash occurs during garbage collection. A garbage collection algorithm satisfying this requirement is called an *atomic garbage collector*. Devising an algorithm for atomic garbage collection requires two interrelated algorithms: an algorithm for garbage collection and an algorithm for recovery during garbage collection.

This chapter presents a method for atomic garbage collection that allows recovery using virtual memory in the event of a soft crash; the next chapter integrates the method into a recovery system for Argus. The description of the atomic garbage collector includes back-

ground and assumptions, a general outline of steps for making garbage collection atomic, a brief review of copying garbage collection, a demonstration that a copying garbage collector is not resistant to crashes, a description of the modifications to a copying garbage collector to make it atomic, and an evaluation of the algorithm.

3.1 Background and Assumptions

Garbage collection techniques can be classified according to whether they are real time or "stop the world." A real time garbage collector works in parallel with the program actually running. Steps of the garbage collection are interleaved with program steps. A "stop the world" garbage collector is invoked by a program when it needs to reclaim storage in its heap. While a "stop the world" garbage collector works, the program is suspended. The user of a system employing "stop the world" techniques usually notices a delay during garbage collection.

The recovery methods developed in this thesis are designed to be used on a standard architecture, a general purpose register machine with virtual memory. Such a machine would have no special purpose hardware to aid in garbage collection. In general, real-time garbage techniques are not suitable for such a machine. Real-time garbage collection techniques need special purpose hardware such as that found on the Lisp Machine [20] in order to be feasible. Thus, this thesis limits itself to a discussion of "stop the world" techniques.

In fact, only copying garbage collection [19,8,4] is considered. Other techniques that can be classified as mark, sweep and compact are not as well suited to virtual memory systems [5]. They require that the accessible objects be traversed more times than is needed for copying garbage collection. Each traversal of the accessible objects means paging overhead. For example, the garbage collector currently used by Argus, similar to the Lisp 2 collector[12], requires four traversals of the accessible objects. Two of these traversals page virtual memory randomly – the mark phase follows the graph of accessible objects, and the phase that recalculates the values of pointers references old object locations in order to find new object locations. Copying garbage collection traverses the accessible objects only twice – once following the graph of accessible objects in from-space and once traversing to-space

sequentially.

It is assumed that the operating system provides primitives that give the implementor of atomic garbage collection control over the paging of virtual memory. Primitives are needed to pin and unpin pages of virtual memory, and to tell the system to write a specific page of virtual memory to the backing store. Pinning a page of virtual memory in physical memory means that the page cannot be written to its place on the backing store until it is unpinned. Pinning primitives are used for buffer management by database systems[10] and in other transaction systems that tie recovery to virtual memory[7,24].

The last assumptions that need to be discussed concern the structure of objects. Objects consist of a *descriptor* and a *body*. The descriptor identifies the type and the length of the object; and the body contains the object's value including pointers to other objects. One assumption is that the descriptor is large enough to contain a pointer. A second assumption is that it is possible to tell the difference between a valid descriptor value and a pointer. This is possible if one bit position of each memory word is used to distinguish pointers from all other values. These assumptions are standard for most systems with variable-sized objects implemented on standard architectures; in particular they hold for the current Argus implementation.

3.2 Outline of Steps for Atomic Garbage Collection

Here is an outline of steps taken at the time of garbage collection to make "stop the world" garbage collection atomic:

1. Record on a medium that survives soft crashes that garbage collection has begun. In the event of a crash, this notifies the recovery system that it needs to use its special algorithm for recovery during garbage collection. This record must be physically recorded on a medium that will survive a soft crash before the garbage collection begins.

2. Garbage collect.

3. Record on a medium that survives soft crashes bookkeeping information about the compacted memory specific to the recovery algorithm used by the recovery system. The information relates virtual addresses before the garbage collection to the corresponding addresses after the garbage collection. In the worst case a pair of addresses might be required for every object accessible from the stable variables. The amount of information required depends on the recovery system and the garbage collection

algorithm. Bookkeeping for a specific recovery algorithm is presented in the next chapter.

4. Write all dirty pages of virtual memory to the backing store to insure that the backing store is consistent, i.e., that all objects are at their new locations.

5. Record on the same medium used in the first step that garbage collection has completed.

A soft crash occurring after step 1, but before step 5 is complete, is a crash during garbage collection.

The simplest method for atomic garbage collection turns a crash during garbage collection into a hard crash. Then any algorithm could be used for garbage collection. Although, such a method is simple to implement, the relative cost of recovery from a crash during garbage collection compared to the cost of recovery at other times would be high. Since soft crashes are much more frequent than hard crashes, the recovery system should be tuned to make recovery from soft crashes as short as possible. The overall cost for recovery would depend on the fraction of time spent garbage collecting. If that fraction were low enough, then the simple method might be acceptable. Otherwise, an atomic garbage collector that allows recovery using virtual memory in the event of a soft crash needs to be devised. For such a collector to be viable, the extra costs it imposes for garbage collection need to be kept to a minimum. After reviewing copying garbage collection, the remainder of this chapter will deal with such a collector.

3.3 Copying Garbage Collection

In a system that uses copying garbage collection, the address space is divided into two semispaces: *from-space* and *to-space*[8,4]. The program allocates all new objects in from-space until the memory in from-space is exhausted or the paging behavior of the program needs to be improved. At that point, garbage collection is initiated.

During garbage collection, all accessible objects are copied from from-space to to-space. As each object is copied, a forwarding pointer is left in its from-space copy. The purpose of forwarding pointers is to preserve sharing in the object structure. Forwarding pointers also prevent an object from being copied more than once into to-space.

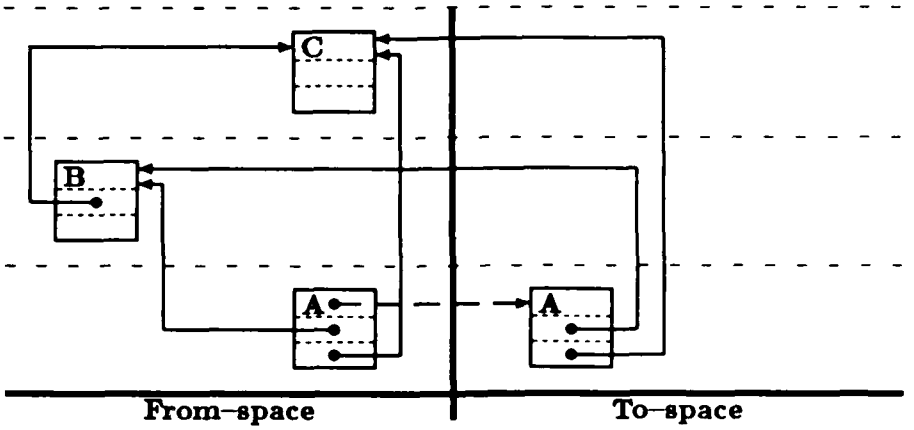
First the root objects are copied to to-space. Then to-space is scanned sequentially for pointers into from-space. As each such pointer is found, it is dereferenced to find the object to which it points in from-space. If the from-space object has a forwarding pointer, that object has already been copied to to-space, so the pointer in to-space is changed to point to the to-space copy. If there is no forwarding pointer, the object has to be copied from from-space to to-space, leaving a forwarding pointer in the from-space copy. Garbage collection ends when all of to-space has been scanned; at that point all of the accessible objects have been copied to and compacted in to-space. Then the roles of from-space and to-space are reversed and the program proceeds.

An example of copying garbage collection can be seen in figure 3.1. Object A, the root object, is copied to to-space. A forwarding pointer is placed in the from-space copy of object A. To-space is scanned sequentially for pointers into from-space. Pointers to objects B and C are found in object A. Objects B and C are copied to to-space to the next free locations. A pointer to object C is found in object B. The forwarding pointer in object C indicates that it has already been copied.

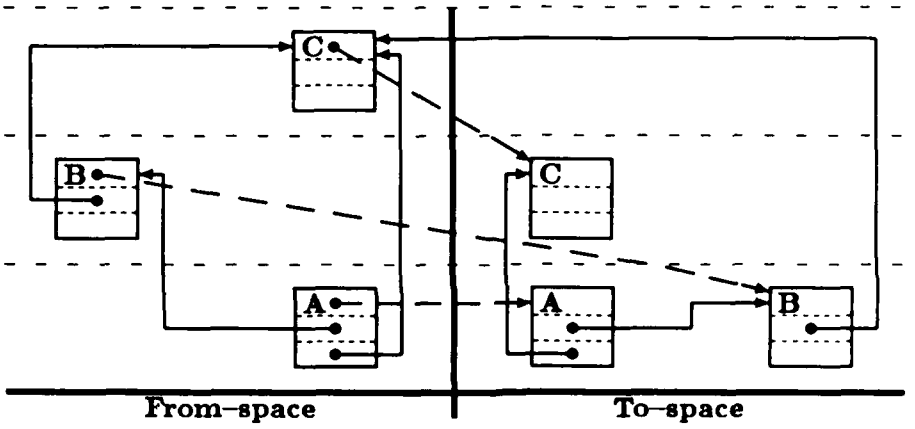
3.4 Crashes

To understand why copying garbage collection is not atomic, consider what happens if a crash occurs in the middle of garbage collection. The pages of virtual memory are being paged in and out of the physical memory according to the way the graph of accessible objects is being traced and copied. A crash will likely leave the contents of the backing store of the virtual memory in an inconsistent state. The following two examples show the two kinds of problems that can occur as a result of a crash in the midst of garbage collection and that need to be prevented by an atomic garbage collector.

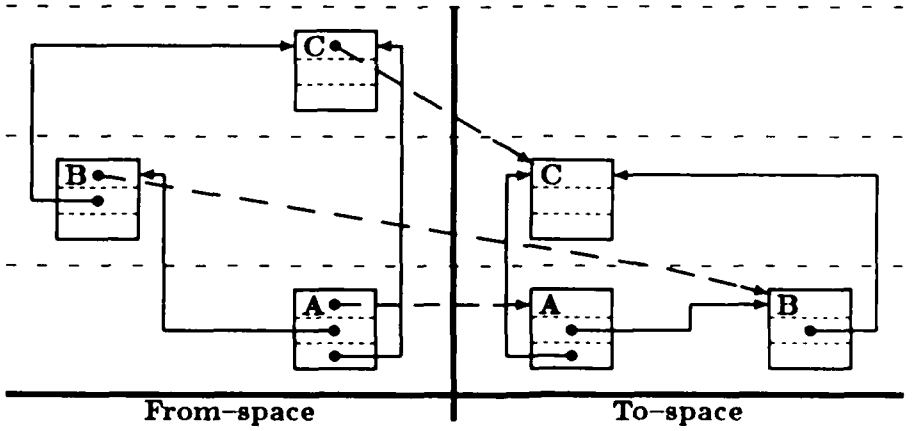
Figure 3.2 shows an example in which object descriptors are lost. Figure 3.2.a shows an object copied from from-space to to-space; a forwarding pointer was placed in the from-space copy. The forwarding pointer takes the place of the object descriptor. The page of from-space on which the old object version resides is written to the backing store. Figure 3.2.b shows what happens if the system crashes before the new version in to-space reaches the backing store. The backing store will not contain a valid version of the object after the crash.



3.1.a: Root Object is Copied

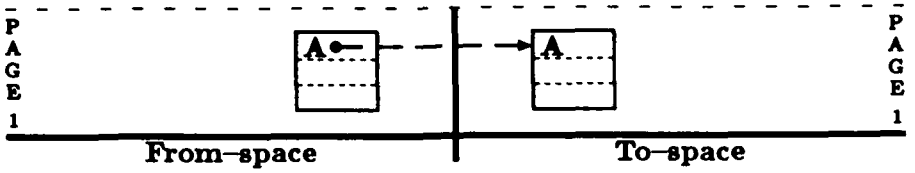


3.1.b: To-space is Scanned

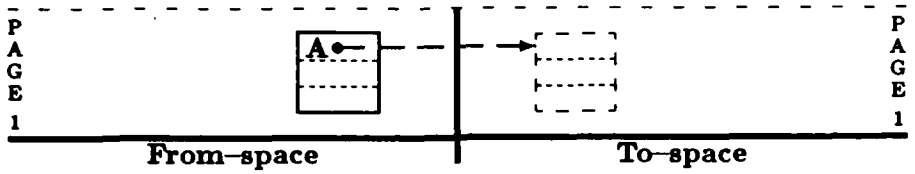


3.1.c: Forwarding Pointers Preserve Sharing

Figure 3.1: Example of Copying Garbage Collection



3.2.a: Virtual Memory Just Before Crash



3.2.b: Backing Store After Crash

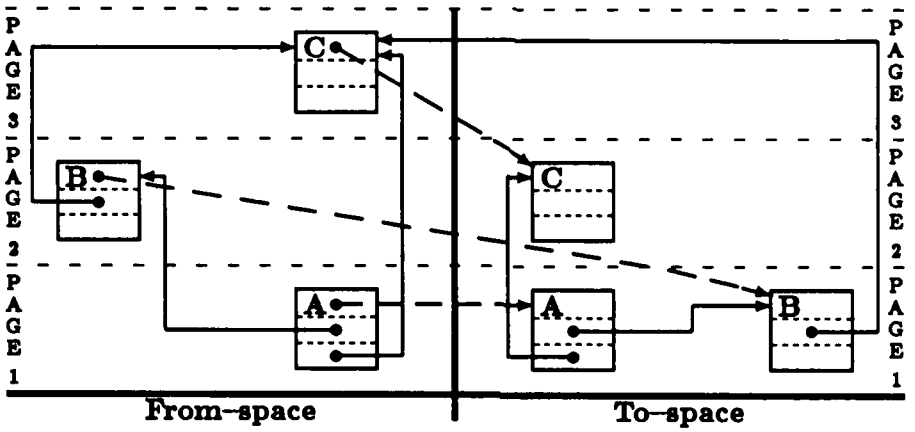
Figure 3.2: Lost Object Descriptor

The object descriptor of the object has been overwritten with the forwarding pointer, and is not available on the backing store for from-space. Neither is it available on the backing store for to-space.

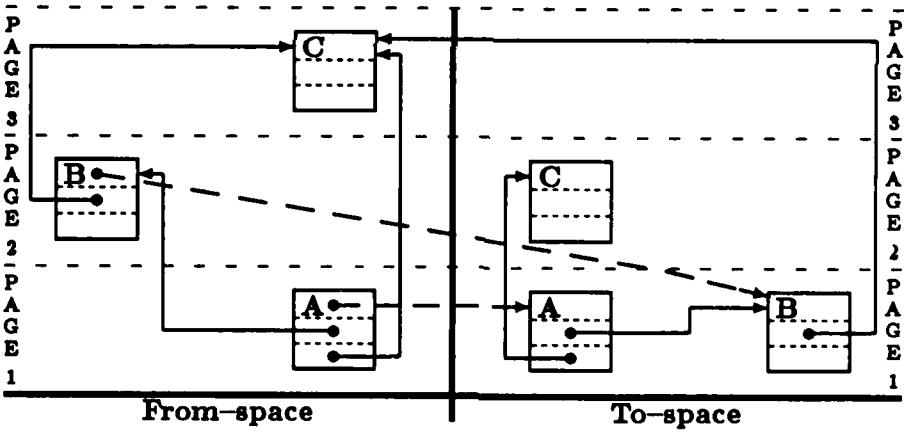
The second example, illustrated in figure 3.3, shows why the pointers on the backing store for to-space cannot be used for recovery after a crash. An object is copied, but its forwarding pointer does not survive the crash; recovering on the basis of information already copied to to-space would not preserve the sharing in the graph of accessible objects. Objects A, B, and C have been copied to to-space, but the pointer to object C from object B has not yet been replaced by a to-space pointer. The crash occurs after pages 1 and 2 of from-space and pages 1 and 2 of to-space have been written to the backing store, but before page 3 of from-space has been written. The result is pictured in figure 3.3.b. The forwarding pointer for object C has been lost even though the object has already been copied to to-space.

3.5 Making Copying Garbage Collection Atomic

A progression of ideas that can be used to make copying garbage collection atomic is presented in order to facilitate the explanation of the actual algorithm. This series of ideas also makes the correctness of the final algorithm apparent. The basic observation motivating all the ideas is that from-space needs to be restored to its original state before the onset



3.3.a: Virtual Memory Just Before Crash



3.3.b: Backing Store After Crash

Figure 3.3: Lost Forwarding Pointer

of garbage collection. Then, after recovery, garbage collection can be restarted using a fresh copy of to-space. Since the object descriptors are the only information overwritten in from-space during garbage collection, reconstructing the object descriptors in from-space is sufficient to allow recovery.

The first idea is to allocate an extra cell per object to hold the forwarding pointer in order that the forwarding pointer need not overwrite any information in the descriptor. Then if a crash occurred during garbage collection, no information would be lost. The backing store for to-space could be discarded and copying garbage collection re-started to a fresh copy of to-space.

The cost of this method is that an extra cell is needed for every object even though the stable objects are the only ones that need to be recovered after a crash. Virtual memory contains both volatile and stable objects. Since a volatile object can become stable at any time by being made accessible from a stable object, volatile objects also need the extra cell.

The next idea is motivated by the inadequacies of the initial idea. It consists of two parts.

First, the stable objects that have to survive crashes can be copied before the volatile objects are copied. This is easy to achieve in Argus since the set of stable objects is defined as all objects accessible from the stable root. Then it is possible to insure that the overhead for making garbage collection atomic is paid only while the stable objects are being copied.

Second, a write-ahead log[9] can be used to record changes to from-space while the stable objects are being copied. The only changes being made to from-space are the insertions of forwarding pointers in place of object descriptors. Only enough information needs to be recorded in the write-ahead log to undo those changes. For each object copied, a pair of values is entered in the log, the first giving a from-space address of the descriptor and the second, the original contents of the descriptor. The write-ahead log principle requires that the from-space page on which the object descriptor resides needs to be pinned in physical memory until the log record recording the change is written to the log. The write-ahead log need not be recorded on stable storage since its storage need not be any more fault-tolerant than the non-volatile memory used for the backing store.

To recover from a crash during garbage collection, the backing store for to-space is

discarded. The write-ahead log is read and as each of its records is processed the information is used to restore the object descriptors in from-space. When the write-ahead log has been processed entirely, from-space will be in the same state as it was before the garbage collection commenced. All dirty pages of from-space are written to the backing store, and the garbage collection algorithm can be restarted from the beginning.

The advantage of the write-ahead log is that the extra price paid for garbage collection is proportional to the number of stable objects. Furthermore the extra storage required is not part of the virtual address space and it is only needed during the garbage collection process. However, extra storage is still required.

The final idea reduces the extra storage required to make copying garbage atomic. It is based on the following observation: *to-space can be used as a write-ahead log for from-space.* The details are presented below.

3.6 Atomic Copying Garbage Collection

Now the algorithm for atomic copying garbage collection can be described. The basic step of a copying garbage collector is the copying of an object from from-space to to-space. This is the step in which atomic copying garbage collection differs from plain copying garbage collection.

3.6.1 Copying Step

The copying step will now be described in detail. First, the page in from-space on which the object to be copied resides is pinned in physical memory. Then, the object is copied to the next available block of storage in to-space and a forwarding pointer is put in the object in from-space, overwriting its descriptor. As each page in to-space fills with copied objects, it is written to the backing store. The from-space page of a copied object is unpinned after the to-space page to which the object was copied reaches the backing store. Figure 3.4 shows the copying step pictorially.

The copying step uses the write-ahead log principle; to-space is being used as a write-ahead log for from-space. In terms of the problems introduced by a crash during garbage collection, the pinning of the from-space page in the copying step prevents the problem

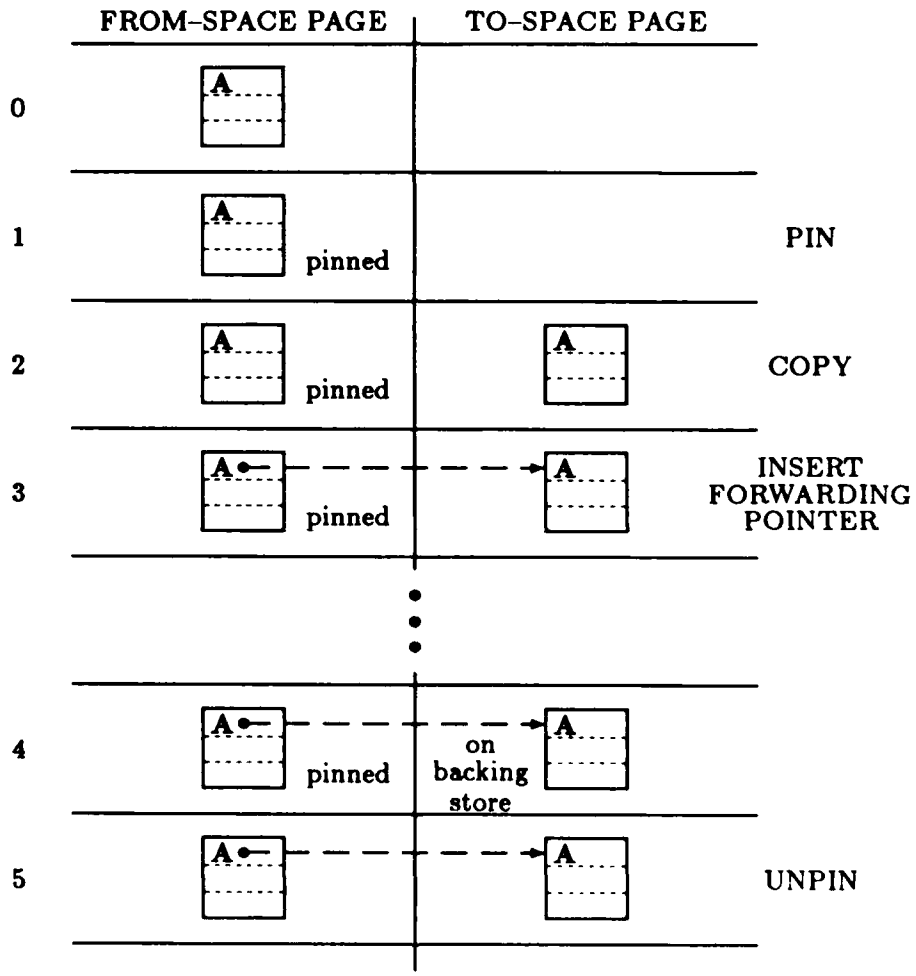


Figure 3.4: Sub-steps of the Copying Step of the Atomic Algorithm

of lost object descriptors. It guarantees that there is always a valid copy of an object's descriptor on the backing store.

Now it is clear why the assumption that object descriptors be big enough to hold a forwarding pointer is needed. The forwarding pointer cannot overwrite any memory location in the from-space copy of the object; it can only overwrite a location whose contents could be recovered from to-space after a crash. Because object descriptors are not changed in to-space by the copying garbage collection algorithm after they are copied, they are an appropriate place for the forwarding pointers.

3.6.2 The Algorithm

Here is the full algorithm. It follows the basic outline for an atomic garbage collector described earlier. The steps that differ from the basic method are discussed below.

1. Write all dirty pages of from-space on which stable objects reside to the backing store.
2. Record on a medium that survives soft crashes that garbage collection is in progress.
3. Use copying garbage collection with the following modifications:
   - The stable root is copied first.
   - To-space is used as a write-ahead log for from-space as described above.
   - To-space pages are written to the backing store as they fill with copied objects.
4. Write all dirty pages of to-space to the backing store.
5. Record bookkeeping information required by the recovery system.
6. Record on the same medium used in the second step that garbage collection has completed.
7. Copy the volatile objects to to-space using normal copying garbage collection.

The first step insures that all stable objects are in a consistent state on the backing store for from-space at the outset of garbage collection. It reduces the interaction between the recovery procedure for a crash during garbage collection and the normal recovery procedure for a soft crash (presented in the next chapter). If it were omitted, the recovery system would need to reconstruct the portions of from-space that had not reached the backing store before the beginning of the garbage collection before it could undo the damage caused by the garbage collection. This could be done using the recovery procedure for a soft crash

to recover the stable state in from-space. As each object is accessed during that procedure its object header would have to be reconstructed by following its forwarding pointer to to-space.

The first step also allows an algorithm for recovery from a crash during garbage collection that is more space efficient. The recovery algorithm, presented in the next section, depends on the fact that the to-space that would have been constructed by the copying process if a crash had not occurred is identical to the to-space that is reconstructed by the recovery algorithm. If the step were omitted, a mutex version recovered from stable storage would not necessarily be the same as the version in from-space before the crash. The reconstructed object graph would be different than the one when garbage collection commenced. Garbage collecting the reconstructed from-space would not yield the same to-space that would have been constructed before the crash.

The actual copying algorithm has already been discussed.

Flushing all dirty pages of to-space to the backing store assures that all of the stable state is on the backing store for to-space in a consistent state. Note that flushing is not required for dirty from-space pages; from-space pages are *not needed* by the recovery system once it has been recorded that the garbage collection is complete.

Only the garbage collection of the stable state needs to be atomic. Once all the stable state has been copied to to-space, all dirty pages of to-space have been flushed to the backing store, and the bookkeeping information has been recorded, the garbage collection has ended as far as the recovery system is concerned. Normal copying garbage collection can be used to complete the copying of the volatile state.

3.7 Crash Recovery

Before discussing the actual recovery algorithm in the event of a crash during garbage collection, a few observations are in order. What can be said about the backing store that survives a crash? From-space is intact except for object descriptors that have been overwritten by forwarding pointers. Because the write-ahead log protocol was followed before pages of from-space were written to the backing store, the object descriptors that have been overwritten in from-space exist in to-space.

The simplest recovery algorithm uses to-space as an undo log. It traverses the stable state in from-space starting with the stable root. Every time a forwarding pointer is encountered in the place of a descriptor, it is dereferenced to retrieve the descriptor from to-space and the original descriptor is returned to from-space. When the whole stable state has been traversed, all dirty pages of from-space are written to the backing store, the storage for to-space is released, and the garbage collection is restarted. This solution requires the stable state to be traversed twice, once to restore the descriptors and once to garbage collect. Alternatively, garbage collection and the restoration of object descriptors can be carried out in parallel if objects are copied to a fresh copy of to-space.

The observation that copying garbage collection is deterministic is the key to designing a more efficient algorithm. The from-space reconstructed using to-space as an undo log is identical to the from-space before the beginning of the garbage collection preceding the crash. Determinism means that starting with two identical copies of from-space as input to the copying algorithm, two executions of the algorithm produce identical copies of to-space. This means that the restoration of object descriptors and the copying algorithm can be carried out in parallel and that the same copy of to-space can be reused. As each object is copied to to-space during recovery, it is copied to the same location to which it was copied or would have been copied before the crash. If the object's descriptor was overwritten with a forwarding pointer before the crash, its original descriptor is found in to-space and restored to the object.

Now the algorithm for recovery from a soft crash during garbage collection can be presented. First, the stable root is reconstructed in to-space. It is reconstructed on the basis of the contents of the descriptor cell of its from-space copy according to the method described below. Then to-space is scanned sequentially as in atomic garbage collection for pointers into from-space. The scan starts with the reconstructed copy of the stable root. As each pointer to from-space is processed, the object to which it points is reconstructed. It, too, is reconstructed on the basis of the contents of the descriptor in its from-space copy. Recovery of the stable state is complete when all of to-space has been scanned.

When processing the pointer to the stable root in from-space or a pointer to an object in from-space encountered during the scan of to-space, the action taken depends on the

contents of the descriptor cell to which the pointer points. There are three cases (only the first two apply to the stable root):

1. The cell is a forwarding pointer to an object that has not yet been reconstructed during recovery. The location of the object in to-space from the original garbage collection and the next location to which the recovery algorithm would copy an object are identical. Thus, the location of the object's descriptor in to-space has been found. The object is reconstructed using its descriptor in to-space and its body from from-space.

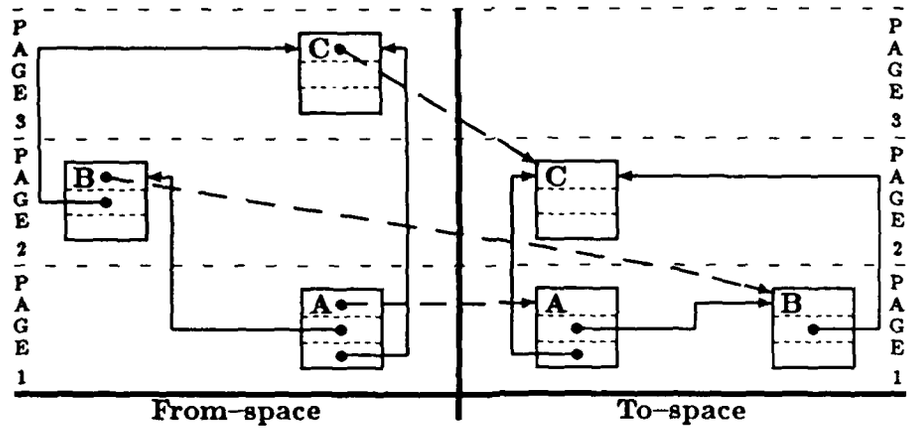
2. The cell is a descriptor. The object's descriptor was not overwritten on the backing store; use the copying step of the atomic garbage collector to copy the object to to-space.

3. The cell is a forwarding pointer to an object that has already been reconstructed during recovery. The forwarding pointer is a valid forwarding pointer; the object has already been reconstructed.

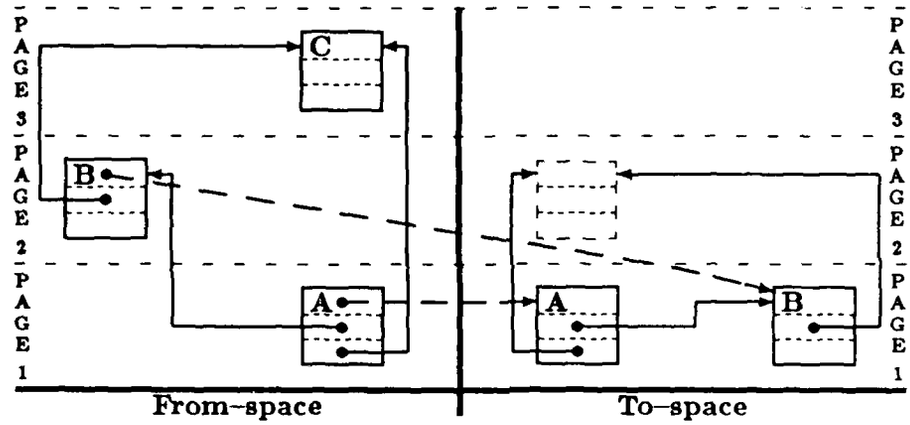
Note that cases 1 and 3 can be differentiated by keeping track of the next free location in to-space during recovery. If the forwarding pointer is less than the next free location (assuming to-space is allocated in the order of increasing addresses), case 3 results. If they are equal, case 1 results.

Figure 3.5 shows an example of recovery after a crash in the middle of garbage collection. Figure 3.5.a shows a possible state for virtual memory before the crash. Figure 3.5.b shows a possible state for the backing store after the crash. The crash occurred after pages 1 and 2 of from-space and page 1 of to-space had been written to the backing store, but before page 3 of from-space or page 2 of to-space had been written. Note that the state pictured in figure 3.5.b is one that could be produced by the atomic copying garbage collector. Both objects whose descriptors have been overwritten with a forwarding pointer in from-space (A and B) have survived in to-space. The forwarding pointer for object C can not have overwritten the descriptor for object C on the backing store for from-space, because object C has not been written to the backing store for to-space.

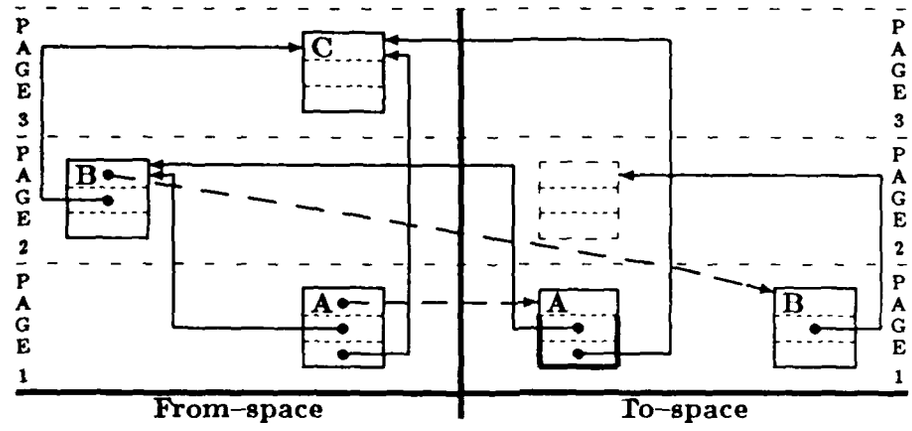
Recovery starts with the reconstruction of the root object in to-space in figure 3.5.c. The root object is an example of case 1 from the algorithm; its descriptor in from-space has a forwarding pointer to the next place to which an object would be copied in to-space (the first location of to-space). The descriptor of the root object is taken from to-space, and its body is recopied from from-space to to-space. For case 1, a tempting "optimization" would



3.5.a: Virtual Memory Before Crash



3.5.b: Backing Store After Crash



3.5.c: Root Object is Reconstructed

be to recover the bodies of objects from to-space. Return to figure 3.5.b and consider what would happen if this were done for the root object. Object C would never be recopied to to-space.

Continuing with figure 3.5.c, to-space is scanned sequentially for pointers into from-space starting with the newly reconstructed root object. The first such pointer is a pointer to object B in from-space. Object B is another example of case 1 from the algorithm; its descriptor in from-space has a forwarding pointer to the next place to which an object would be copied in to-space. Thus, the object has not yet been reconstructed during recovery. It is reconstructed in to-space using the descriptor from to-space and its body from from-space. The result is pictured in figure 3.5.d.

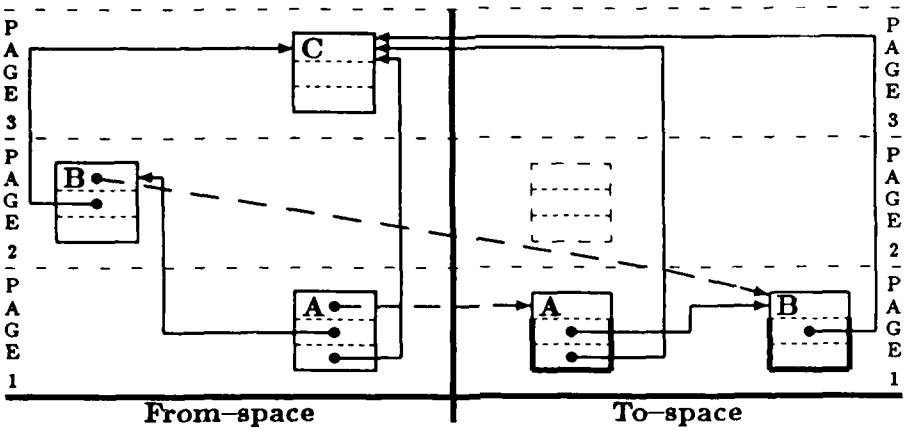
Continuing the scan of to-space for pointers into from-space in figure 3.5.d, a pointer to object C is found. Object C is an example of case 2 from the algorithm; the descriptor for object C is on the backing store for from-space. It is copied as is from from-space to to-space. The result is pictured in figure 3.5.e.

The next pointer into from-space from to-space is the pointer to object C in object B. This is an example of case 3. Object C is an object that has already been reconstructed by the recovery algorithm; the forwarding pointer in from-space points to an area of to-space that has already been reconstructed. The result is pictured in figure 3.5.f.

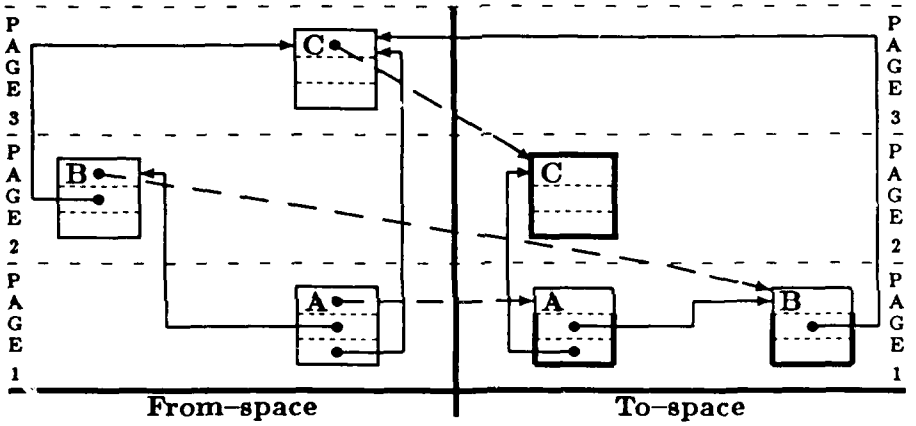
3.8 Evaluation of the Algorithm

The algorithm presented for atomic garbage collection should be compared to the normal copying algorithm to evaluate what costs it adds. The atomic algorithm only adds cost to the garbage collection of stable objects. There is no extra cost for the garbage collection of volatile objects. Since only stable objects need to survive a crash this is a desirable property.

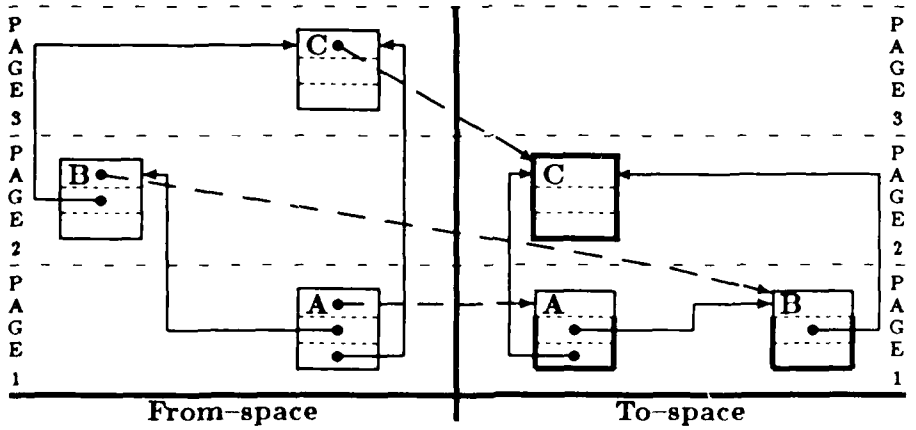
The added costs are proportional to the number of stable objects. In the worst case, every object being copied resides on a page of from-space that has not yet been pinned, and there is one page pinned (and unpinned) for every object copied. Pin and unpin operations should be fast. A possible implementation follows. The pin operation sets a bit in the page or frame table; the unpin operation resets the bit. Keeping track of the pinning is a little



3.5.d: Case 1



3.5.e: Case 2



3.5.f: Case 3

Figure 3.5: Recovery Example

more complicated. The atomic garbage collector keeps a list of pinned from-space pages for each to-space page to which objects are being copied. To avoid conflicts of unpinning a page of from-space when it is pinned on behalf of more than one to-space page, it also keeps a counter for each pinned from-space page that counts the number of to-space pages for which the from-space page has been pinned. The atomic garbage collector calls a primitive of the virtual memory system to notify the system that a to-space page has to be written to the backing store. The virtual memory system notifies the collector when the page has actually been written by setting a flag. The collector checks the flag when it is ready to write the next page of to-space to the backing store. The collector decrements by one the counter for each from-space page in the list for a to-space page that has reached the backing store. If the counter is 0, the from-space page is unpinned.

The atomic garbage collection algorithm increases I/O activity to the backing store. Pages of to-space are almost always written to the backing store twice – once when they fill with objects copied from from-space and the second time when the whole page has been scanned for pointers to from-space. The original non-atomic algorithm could exhibit the same behavior, but a smart paging policy for garbage collection might avoid it. The original non-atomic algorithm does not need to use large areas of physical memory for pinned from-space pages; more pages of physical memory are available to be allocated to pages of to-space. Therefore, using the original algorithm it is more likely that a to-space page could stay in physical memory from the time the first object is copied to it until its last location has been scanned for a pointer to from-space.

Although, there is also a potential for more faults on pages of to-space using the atomic algorithm because more pages of physical memory have to be devoted to from-space pages, buffering techniques could be used to avoid processing delays caused by page faults. The order in which the pages of to-space will be scanned for pointers into from-space is sequential. I/O activity and copying can proceed in parallel; while one page of to-space is being scanned, the next page can be read into physical memory.

For acceptable performance, the physical memory should be large enough to avoid increasing I/O activity by writing a page of to-space to the backing store more than twice. If the number of pages pinned at once on behalf of one to-space page is too high, then a page

of to-space might have to be written to backing store before it is full. A rough approximation shows that the number of pages of from-space that are pinned at once is reasonable. Assuming that pages are 1024 bytes long and that the average object in Argus is 5 words long (20 bytes); there are 51 objects per page on average. The average number of pages that need to be pinned at once on behalf of one to-space page should be much lower than 51 due to locality of reference.

3.9 Extensions of the Algorithm

Several authors[6,20] have suggested that copying garbage collection can be used to increase the locality of refence in virtual memory for the objects it copies. The copying algorithm as presented in this chapter traverses and copies the graph of accessible objects in a breadth-first fashion. Moon[20] argues that, for Lisp, depth-first copying yields better locality than breadth-first copying. He suggests a method of copying called approximate depth-first copying, which uses depth-first copying on partially filled pages of to-space. A similar strategy for preserving locality of atomic objects could also be used in the implementation of Argus.

Proposals for changing the order in which objects are copied for the purpose of increasing locality of reference can be incorporated into atomic copying garbage collection as long as the resulting algorithm is deterministic. The recovery algorithm would have to be changed so that it copies objects in the same order as the new garbage collection algorithm. The recovery algorithm works as long as any two executions of the new copying algorithm always copy an object in from-space to the same place in to-space.

Chapter 4

Recovery Using Virtual Memory

This chapter outlines a method for recovery using virtual memory. First recovery for built-in objects is discussed. The topics include the representation of built-in atomic objects, the log, checkpoints, updating object headers, restoring built-in atomic objects after a crash, garbage collection and a sketch of the recovery algorithm. Next, the recovery of mutex objects is covered. Finally, methods for housekeeping the log are discussed.

Where there are a number of solutions possible to solve a particular problem, a single solution has been chosen for the method being outlined. Chapter 5 discusses alternatives.

When a crash is discussed in this chapter, this means a soft crash unless otherwise indicated.

4.1 Representation of Built-in Atomic Objects

Recovery using virtual memory restricts the representations possible for built-in atomic objects. The representation chosen must ensure that there is information on the backing store that is usable after a crash. A shadowing mechanism similar to the one used in the current implementation of Argus can fulfill this requirement[18].

In the description that follows some details of the representation have been simplified. Information required to support nested actions is not discussed; it does not need to survive a crash because actions in progress that have not prepared are aborted by a crash. Therefore, the representation chosen for that information is not constrained by the recovery system.

The shadowing mechanism is implemented by using a header for each built-in atomic object through which all references to the object are directed. The header contains a

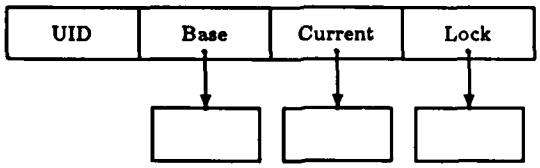


Figure 4.1: Built-in Atomic Object

unique object identifier (*UID*), pointers to a ***base version*** and a ***current version***, and lock information. The *UID* is required to maintain inter-object references on stable storage. The ***base version*** is the last object version to be committed by an action; it is not modified by actions in progress. Modifications to the object are made to the ***current version***. The lock information is associated with the current version; it consists of the AID, or action identifier of the action holding a lock on the object and whether the lock is read or write. The updating of pointers in the header is closely coordinated with two-phase commit for a top action that holds a write lock on the object. The pointer to the base version is updated only as part of the commit phase during which the current version replaces the base version.

The representation for a built-in atomic object is illustrated in figure 4.1. The header and each component of the built-in atomic object to which a field in the header points are themselves objects.

Atomic object headers do not move within the virtual address space except during garbage collection. Garbage collection is also the only way the pointers in the header to a base version or prepared version can change outside of the two-phase commit protocol of an action.

Atomic object headers and the components of the object to which fields of the header point can span page boundaries. The only restriction is that individual fields of the header not span page boundaries.

4.2 The Log

As in the current recovery system[21,22], stable storage for a guardian is organized as a log. The structure of the log and the algorithm for writing to the log remain essentially the same. This section discusses the way resilient objects are referenced in the log, the way objects are copied to the log, and the way log records are formatted and written to the log.

4.2.1 References to Resilient Objects

References to resilient objects are recorded in the log using <UID, virtual address of object header> pairs. The UID uniquely identifies the object for recovery from a hard crash. The virtual address of the object header is used both to locate the object in virtual memory and to identify the object for recovery from a soft crash. Recording both the UID and the virtual address might seem redundant. However, objects move during garbage collection. By pairing UIDs and virtual addresses in every reference to an object, a map of UIDs to virtual addresses of object headers is being recorded incrementally in the log; chapter 5 discusses an alternative.

4.2.2 Copying an Object

The method used to copy most resilient objects to the log is incremental. Specifically, each built-in atomic object is copied to the log in a separate incremental step. Immutable objects such as sequences may or may not be copied incrementally depending on the implementation[1]. Inter-object references between objects that have been copied in separate incremental steps are supported using resilient object references, <UID, virtual address of object header> pairs.

Objects in virtual memory are stored in a heap. When one object contains a second object, the second object is not physically contained within the first object; rather, a pointer is used to refer to it. Exceptions are made for small objects such as integers and characters for efficiency reasons. When an object is copied to the log, it is flattened or linearized with respect to its representation in the heap. Specifically, when the current version of a built-in atomic object is copied to the log, its log version is contiguous and contains the current version followed by contained objects that are being copied in the same incremental step (for example, sequences). Pointers to other objects copied in a separate incremental step are replaced by resilient object references in the log version. Pointers to objects copied in the same step are replaced by relative offsets within the log version.

[1]The recovery system can decide whether or not to copy an immutable object incrementally depending on its size and the degree to which it is shared. If it is large (eg. a large sequence) and is shared by more than one stable object, it saves space in the log to copy it just once.

4.2.3 Log Records and Writing to the Log

There are two types of log records: *data records* and *outcome records*. A *data record* contains the value for a single resilient object. That value is flattened and copied to the log as discussed previously. *Outcome records* are used to record information during two-phase commit. Figure 4.2 illustrates the contents of these records. There are two categories of outcome records: those containing information for participants in two-phase commit and those that contain information for coordinators. The former category includes the *prepared*, *committed*, *aborted*, *prepared_data* and *base committed* records; the latter includes the *committing* and *done* records. The former category also includes *mutex* records; section 4.8 discusses mutex objects and records.

Each outcome record has a log pointer as a component, linking it to the previous outcome record in the log. This reverse chain of log pointers is followed during recovery to determine the order in which log records are processed. The outcome records form a log within the log.

Writing to the log is coordinated with two-phase commit. When an action prepares, the recovery system writes a data record to the log for each resilient object that was modified by the action and is accessible from a stable variable. The data record contains the value of the object's current version. The system determines which objects need to be written by keeping track of the set of objects that were modified by the action (Modified Object Set or MOS) and intersecting that set with the set of objects accessible from the stable variables (Accessibility Set or AS).

A base_committed record is written to the log for any object that was made accessible from the stable variables by the action (called a newly accessible object). The base_committed record contains the object's UID, the virtual address of its header, and an object value containing the value of the object's base version in the same format as a data record.

If the action also modified the newly accessible object, its current version is written to the log as a separate data record. A prepared_data record is written to the log for any newly accessible object that is in the MOS of another action that has already prepared but not committed. The prepared_data record contains the object's UID, the virtual address of its header, the action identifier of the action that had previously prepared, and an object value

4.2.a: Data Entry

object value

4.2.b: Outcome Entries for Participants

prepared

<virtual address, UID, log pointer>
...
action identifier
log pointer

committed

action identifier
log pointer

aborted

action identifier
log pointer

base_committed

UID
virtual address of object header
object value
log pointer

prepared_data

UID
virtual address of object header
action identifier
object value
log pointer

4.2.c: Outcome Entries for Coordinators

committing

guardian identifier
...
action identifier
log pointer

done

action identifier
log pointer

Figure 4.2: Format of Log Entries

containing the value of the object's current version in the same format as a data record. Note that the prepared_data and base_committed records are hybrid records containing both outcome information and object values.

Once all of the data records, base_committed, and prepared_data records have been written to the log on behalf of a preparing action, the prepared record is forced to the log. The prepared record contains the action identifier for the action and a list of triples, one triple for each resilient object the action has modified. Each triple contains the UID of the object, the virtual address of the object's header in virtual memory and a log pointer to the data record containing the value of the object's current version.

When a guardian receives the commit message from the coordinator of two-phase commit for an action for which it was a participant, it installs the current versions as new base versions and releases locks for the objects modified by the action, and writes the committed record to the log. If the guardian receives the abort message, it discards the current versions and releases locks for the objects modified by the action, and writes the aborted record to the log. Both the committed and aborted records contain the action's identifier.

When a guardian acting as coordinator for an action has received prepared messages from each of the participants for that action and has decided to commit, it writes the committing record to the log. The committing record contains the identifier for the action and a list of identifiers for the guardians acting as participants. When the coordinator receives an acknowledgement of commit from each of the participants, it writes the done record to the log. The done record contains the identifier of the action.

4.3 Checkpoints

The current recovery system processes the log in its entirety to recover from a crash. The new recovery system proposes to take advantage of information that survives on the backing store after a soft crash and only process part of the log. In order to do so, an inexpensive mechanism is required for insuring that there is information on the backing store that is reliable after a crash and determining what portion of the log has to be processed to recreate the unreliable information. Checkpointing, a well known technique applied in other systems such as System R[10], fulfills these goals.

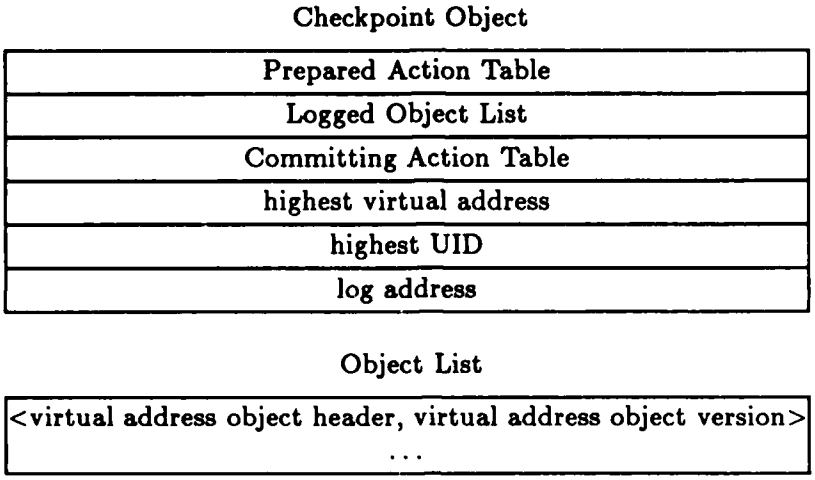


Figure 4.3: Format of Checkpoint Object

A checkpoint is taken periodically by halting computation at the guardian temporarily so that the recovery system is quiescent, writing all dirty pages of physical memory to the backing store, forcing buffered records to the log, and constructing and installing a new checkpoint object in virtual memory. These steps insure that the contents of the stable state on the backing store is consistent with the stable state in the log at the time of the checkpoint.

The remainder of this section discusses the work saved by a checkpoint and the contents of the checkpoint object, the installation of a checkpoint, and quiescence.

4.3.1 Work Saved by a Checkpoint

A checkpoint saves work by writing all dirty pages of physical memory to the backing store and recording information in a checkpoint object. Figure 4.3 shows the format of the checkpoint object.

A checkpoint saves all of the work done by actions that have committed before it is taken. Writing all dirty pages of the physical memory to the backing store insures that the base versions and the pointers to base versions in object headers are on the backing store. A pointer to a base version will be changed only if an action that modifies the object commits after the checkpoint. In that case the base version saved by the checkpoint is superceded. Thus, after a crash, no log records need be read for an action that committed before the

last checkpoint.

A checkpoint also saves the work done by actions that are prepared when it is taken. The checkpoint guarantees that the current versions for objects that have been modified by prepared actions will be on the backing store. However, when the prepared actions commit after the checkpoint, the object headers are modified and the pointers to the current versions are lost. If the addresses of the current versions for all objects modified by prepared actions are included in the checkpoint object, the versions can be found after a crash without restoring them from the log. After a crash, the recovery system needs to determine the actions that were prepared at the time of the checkpoint for which the guardian was a participant and for each such action the list of stable objects that it modified. The information concerning prepared actions is recorded in the Prepared Action Table in the checkpoint object. It contains <action identifier, object list> pairs, one pair for each action that was prepared at the time of the checkpoint. There is an entry in the object list for each stable object modified by the action. Each entry is a <virtual address of object header, virtual address of current version> pair. Note that after a crash, no prepared, prepared_data or data record for a prepared action written to the log before the last checkpoint need be read; all of the required information is included in the Prepared Action Table.

A checkpoint saves some of the work done by actions that are in progress when it is taken. Some actions, though they are not yet prepared at the time of the checkpoint, might have already written data records to the log. The situation arises when using early prepare, for example. Early prepare is a way of reducing the latencies of two-phase commit using an eager strategy – data records are written to the log after a handler call completes while the guardian is otherwise idle in anticipation of the prepare message from the coordinator. Early prepare is described in the papers on the current recovery system in Argus[21,22]. For such cases, the current versions of objects that have already been logged will be written to the backing store during the checkpoint. If the action prepares using these same versions, the versions will survive on the backing store after the crash; only pointers to the versions need to be recovered. The Logged Object List is included in the checkpoint object to allow these object versions to be restored to virtual memory without reading their data records from the log. It contains an entry for each object version which has been written to the

log in a data record on behalf of an action in the process of preparing at the time of the checkpoint. The entry maps the virtual address of the object's header to the virtual address of the object's version that was written to the log.

After a crash the recovery system needs to determine the actions that were committing at the time of the checkpoint for which the guardian was coordinator and the list of participants for each such action. It must do so without scanning the whole the log. The Committing Action Table in the checkpoint object is a table of <action identifier, log address> pairs, one pair for each action that was committing at the time of the checkpoint. The log pointer points to the committing record for the action. Recall that the committing record contains the list of participants.

Both the recovery system and the guardian, once it has recovered, need to allocate memory without overwriting the stable state on the backing store that has survived the crash. If virtual memory is allocated continuously from low to high addresses, then it is sufficient to record the address of the next free location in virtual memory at the time of the checkpoint in the checkpoint object. Other schemes for allocating memory are possible with the restriction that any scheme used must be able to record concisely what memory addresses were in use at the time of the checkpoint. Note that any part of virtual memory allocated after the checkpoint does not contain information that needs to survive a crash.

A guardian needs to continue creating new atomic objects after recovery. Each atomic object is given a unique UID. Argus assigns UIDs by generating them in ascending numerical order. In order to continue generating UIDs after the crash without repetitions, the recovery system needs to determine the highest UID in use at the time of crash. In order to do so without scanning the whole log, the highest UID in use at the time of the checkpoint needs to be recorded in the checkpoint object.

To determine what portion of the log must be read after a crash, the log address of the last record forced to the log at the time of the checkpoint is recorded in the checkpoint object. No record written to the log before the checkpoint is read after a crash, except for records to which the Committing Action Table points. The log address at the time of the checkpoint is also used to determine when the information recorded in the Logged Object List can be used. Data records with a smaller log address contain object versions that are

on the backing store and do not need to be recovered from the log.

More information has been included in the checkpoint object than necessary for recovery. The extra information lessens the number of log records that need to be read after a crash. Recall that during recovery the log will be read in reverse order. The checkpoint object is processed after all the outcome records written to the log after the checkpoint have been processed. The minimum information necessary for the Committing Action Table would have been a list of log addresses of the committing records for committing actions. Including action identifiers means that committing records for actions for which a done record has already been processed will not have to be read from the log. The minimum information required for the Prepared Action Table would have been the list of log addresses of prepared and prepared_data records for prepared actions. However, once the action identifiers and the virtual addresses of current versions are included, all the information in those records has been duplicated and their log addresses are no longer needed.

4.3.2 Installing a Checkpoint

The information recorded in a checkpoint object is used only when recovering from a soft crash. Thus, the checkpoint object can be stored on the backing store of virtual memory; it does not have to be written to stable storage. To find it after a crash, a pointer to the current checkpoint object is kept at a known place in virtual memory. The stable root is already kept at a known place in virtual memory and is the root for the graph of objects that must survive a crash. Hence, it is appropriate to store the pointer to the current checkpoint object in the stable root. As will be seen in a later section, this choice is also good for garbage collection.

Installing the new checkpoint object is necessary for the checkpoint to take effect. Installation has to occur atomically as the last step in taking the checkpoint. A new checkpoint object is constructed and written to the backing store. Then the pointer to the current checkpoint object is changed to point to the new checkpoint object, and the page of virtual memory on which the pointer resides is written to the backing store.

At guardian creation, an initial checkpoint object is created and installed. Its Prepared Action Table, Committing Action Table and Logged Object List are empty. The highest

virtual address, highest UID, and the last log address are assigned their initial values. Thus, if a crash occurs before the first real checkpoint, the whole log will be read and the stable state will be reconstructed in virtual memory from scratch.

4.3.3 Quiescence

A checkpoint can be taken only when the recovery system is quiescent. The condition for quiescence is that no action be in the middle of the commit phase of two-phase commit as a participant. Recall that during the commit phase of two-phase commit the current versions of atomic objects modified by an action are installed as base versions, write locks are released, a commit record is written to the log and a committed message is sent to the coordinator.

Suppose that the condition did not hold and consider the following scenario:

1. Commit record is written to the log, but base versions not yet installed and locks not yet released.
2. Checkpoint.
3. Base versions installed and locks released.
4. Crash before object headers reach backing store.

After the crash, no information will be available about the outcome of the action; it will not be listed as prepared in the checkpoint object and there will be no record of it in the portion of the log written after the checkpoint. The recovery system will have no way of restoring the objects or releasing the locks of an object modified by the action; it does not know whether the action committed or aborted.

Requiring that no action be in the middle of two-phase commit when a checkpoint is taken solves the problem. If there is any action in the middle of the commit phase, it is allowed to finish completing before the checkpoint is taken. This insures that a snapshot of the stable state consistent with information in the log is checkpointed to the backing store.

This condition for quiescence is easily satisfied. Releasing locks and installing current versions as base versions does not require additional memory to be allocated. Hence, no garbage collection could be triggered. At worst, there might be a short delay for page faults during which processing at the guardian has already been halted. Thus, most of the

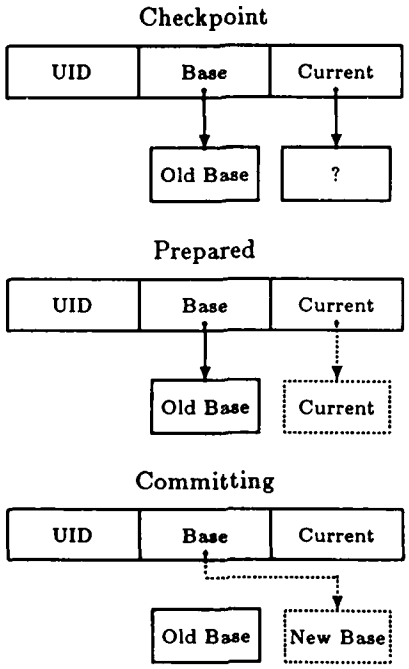


Figure 4.4: Updating an Object Header at Commit

time spent taking a checkpoint is spent flushing the dirty pages of physical memory to the backing store. Overall, checkpoints are cheap.

4.4 Updating Object Headers

Object headers are updated during the commit phase of two-phase commit for a participant in an action. If the steps taken during the commit phase are not ordered correctly, information can be lost by a crash.

Suppose that the new base versions were installed and the locks were released before the commit record were written to the log. The system could crash after the installation of the new base versions, but before the commit record reached the log. Now, consider the case in which the only action to modify an object since the last checkpoint has prepared, but no commit record has been written to the log for it before a crash. Figure 4.4 shows the object's state on the backing store at the checkpoint, when the action prepared, and after the new base version has been installed. In the figure, solid lines show information that is

consistent on the backing store; dotted lines show information that was modified after the checkpoint. After a crash, the base pointer in the object header could point either to the original base version, or to the current version which was being installed as the new base version at the time of the crash. The recovery system cannot distinguish the two cases. If the base pointer points to the new base version, it needs to be restored; the new base version might have been created after the last checkpoint and is not guaranteed to be in a consistent state on the backing store.

Making sure that the commit record is physically in the log before releasing locks or installing new base versions solves the problem. The ambiguities as to whether an action has started to commit are removed. Locks have been released and base versions installed only if there is a commit record in the log for the action[2].

4.5 Scenarios: Restoring Built-in Atomic Objects

This section shows how the information on the backing store of virtual memory is related to information in the log after a crash. The state of a built-in atomic object on the backing store after a crash depends on whether it has been modified by an action since the last checkpoint and, if it has been modified, what stage of two-phase commit had been reached by the modifying action. There are six cases to consider. These are depicted and discussed below.

There are two diagrams for each case. The first diagram pictures a log showing the outcome records written to the log for the last action that modified the object before the crash. The record labeled "Checkpoint" in the diagram is the last record forced to the log before the checkpoint. The second diagram pictures the state of the object on the backing store. Solid lines show information that is on the backing store; dotted lines show information that might have been lost in the crash.

1. The last committed action to modify the object committed before the last checkpoint, and no prepared or committed action has modified the object since the last checkpoint. The object might have been modified by an action that aborted, including the situation in which the modifying action was aborted by the crash. In this case all the necessary information required for the object has survived the crash; both the base version

[2]Note that two-phase commit already requires that the commit record be physically in the log before the committed message is sent to the coordinator.

and the pointer to the base version in the header will be on the backing store. The pointers to the current version and the lock in the header could point to inconsistent information; this information is associated with an aborted action. These fields of the header need to be cleared during recovery.

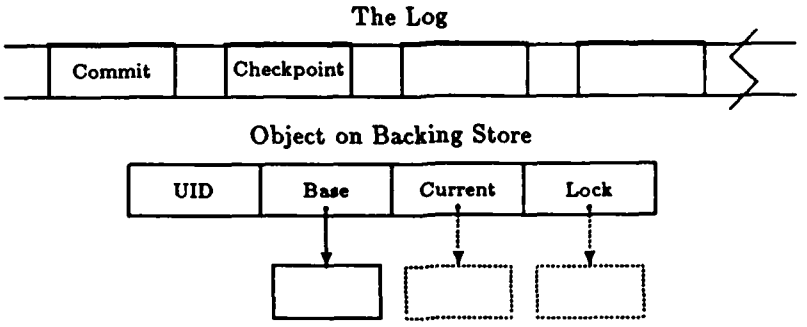


Figure 4.5: Restoring Built-in Atomic Objects: Case 1

2. At least one action that modified the object has both prepared and committed since the last checkpoint. There are no guarantees on the state of the object header on the backing store, or on the state of the base or current versions. The new committed base version and the new current version, if applicable, have to be restored from the log, and the object's header updated to reflect the restoration.

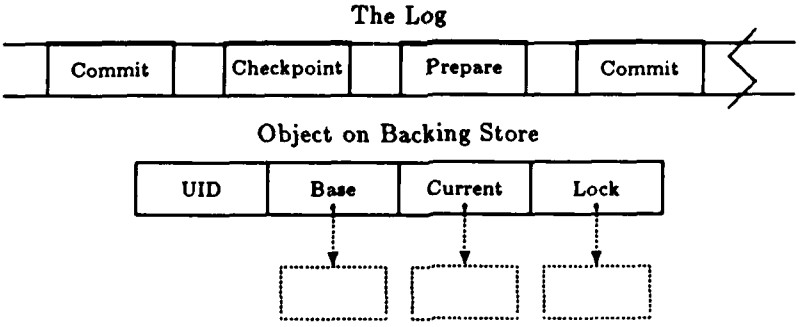


Figure 4.6: Restoring Built-in Atomic Objects: Case 2

3. The last committed action that modified the object prepared before the checkpoint, but committed after the checkpoint. The new base version, which was prepared at the time of the checkpoint, is guaranteed to be on the backing store, but the pointer to it might have been lost. Since there is no way of knowing whether the object's header was written to the backing store since the commit, there is no way to distinguish whether the base version pointer in the object's header points to the old base version or the new base version. There is also no guarantee that the pointer to the current version in the object header still points to the version that had been the current version at the time of checkpoint. After the commit, but before the crash, another

action might have obtained the lock on the object and begun modifying it; the pointer to the current version in the header on the backing store might point to this version after the crash. The object can be recovered by restoring the pointer to the new base version from the checkpoint object and clearing the other fields in the header.

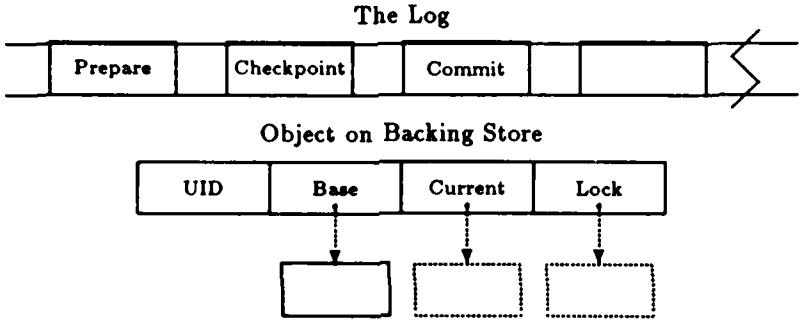


Figure 4.7: Restoring Built-in Atomic Objects: Case 3

4. The last action to modify the object prepared before the last checkpoint, but no committed or aborted record had been written to the log for it before the crash. The checkpoint guarantees that the base and current versions are on the backing store. Because the commit record is forced to the log before headers are updated, it is known that the base pointer in the header has not been modified. However, abort records are not forced to the log. The action might have released the lock in the process of aborting and the abort record never reached the log. In between releasing the lock and the crash, another action might have obtained the lock and created a new current version. The pointer to the current version and the action's lock have to be restored using information in the checkpoint object.

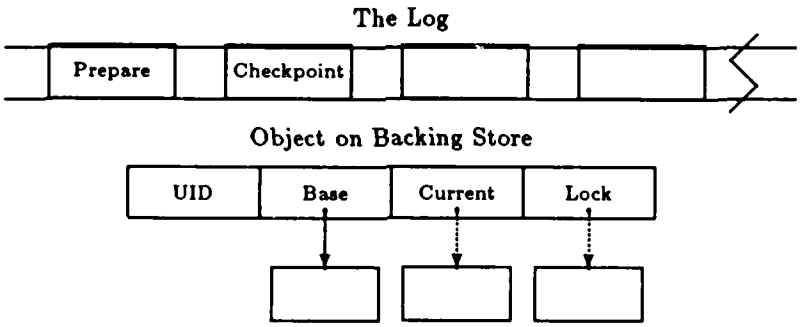


Figure 4.8: Restoring built-in Atomic Objects: Case 4

5. The only action that modified the object since the last checkpoint has prepared but no committed or aborted record has been written to the log for it before the crash. The checkpoint guarantees that the base version is on the backing store. Because the commit record is forced to the log before object headers are updated, it is known that

the base pointer in the header has not been modified. The current version and the pointers to it and its associated lock might have been modified after the checkpoint. They can be restored using the information in the prepared record. Note that a data record for the current version might have been written to the log before the checkpoint. In that case the current version is on the backing store and the pointer to it is restored using information from the checkpoint object.

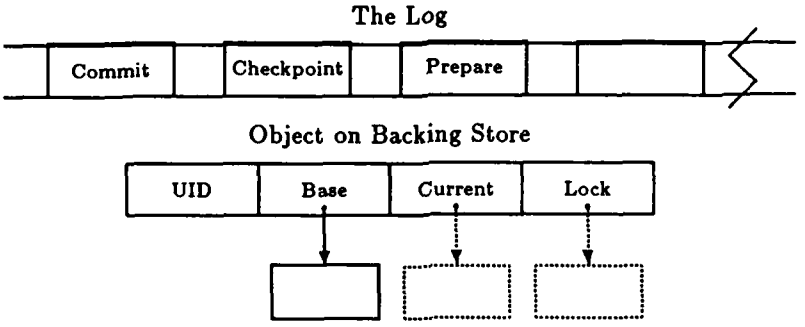


Figure 4.9: Restoring Built-in Atomic Objects: Case 5

6. The object has been logged for the first time after the checkpoint. There are no guarantees about the contents of the object header, or the base or current versions; they have to be restored from the log. The object header itself might have to be recreated. If the virtual address of the object's header is less than the highest virtual address allocated at the time of the checkpoint, then the object header was created before the checkpoint. Otherwise, the object header has to be recreated.

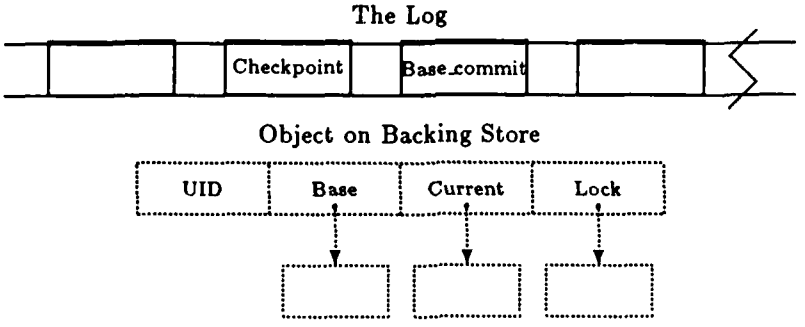


Figure 4.10: Restoring Built-in Atomic Objects: Case 6

4.6 Garbage Collection

A method for atomic garbage collection was presented in the last chapter; it needs to be integrated into the proposed recovery system. The strategy for dealing with "stop the

world" garbage collection is to use two recovery algorithms: the normal algorithm for a soft crash and a special algorithm for a crash during garbage collection. The outline for atomic garbage collection presented in the previous chapter is repeated here with details appropriate to the recovery system inserted.

1. Stop all processing at the guardian; make sure the recovery system is quiescent. The time waiting for quiescence is short and no memory needs to be allocated while waiting.
2. Take a checkpoint. The checkpoint allows the recovery algorithm for atomic garbage collection presented in the last chapter to be used. It also limits the portion of the log that needs to scanned to recover from a crash after garbage collection has completed.
3. Set a garbage collection flag and force it to the backing store. Because the backing store survives a soft crash, virtual memory is an appropriate place for the flag. Like the pointer to the checkpoint object, the flag is kept at a known location in virtual memory. The page on which it resides is forced to the backing store to insure that an indication that garbage collection is in progress survives a crash.
4. Use atomic garbage collection to copy the stable state and the checkpoint object to to-space. Note that the checkpoint object is copied to to-space as part of atomic garbage collection because it is accessible from the stable root. The virtual addresses in it are translated to point to the corresponding objects in to-space by the copying algorithm.
5. Write all dirty pages of to-space to the backing store.
6. Reset the garbage collection flag and force it to the backing store. Note that this has the side effect of installing the to-space version of the checkpoint object. The flag is forced because it must be on the backing store before the first outcome record reaches the log after garbage collection.

If a crash occurs during garbage collection, the recovery algorithm from the previous chapter is used. When it is finished, all dirty pages of to-space are written to the backing store and the garbage collection flag is reset and forced to the backing store.

If a crash occurs after garbage collection, the normal recovery algorithm does not need to do anything special to account for the movement of objects in memory during the garbage collection. Records written to the log after the garbage collection contain the new virtual addresses for the objects to which they refer. The virtual addresses in the checkpoint object were translated to new virtual addresses when it was copied to to-space.

4.7 Recovery of Built-in Atomic Objects

This section describes the algorithm for recovery from a soft crash. The purpose of recovery is to restore to virtual memory all the information required by a guardian to continue normal processing. Included are the stable state, information about actions that were prepared at the time of the crash and information required to create new stable objects. Information for actions that were prepared is required both for actions for which the guardian was a participant and for which the guardian was the coordinator. Knowledge of the highest UID assigned so far to any stable object is required to create new stable objects. The stable state is recovered using a combination of information left on the backing store before the crash and in the log. The other information must be recovered from the log. The method for processing the log is similar to the one used by the current recovery system. The presentation of the recovery algorithm is based on the presentation of that method[22].

When recovering from a soft crash, the recovery system begins by recreating the virtual address space of the guardian using the guardian's backing store from before the crash. Then the garbage collection flag is checked. If garbage collection was in progress, recovery proceeds using the recovery algorithm outlined in the previous chapter. When that algorithm completes, the garbage collection flag is reset. Garbage collection was preceded by a checkpoint and no changes were made to the stable state between that checkpoint and the crash. Thus, no records need to be processed from the log. However, information still has to be reconstructed about actions that were prepared or committing at the time of the crash.

Whether or not garbage collection was in progress, recovery then proceeds by constructing a table, called the CPVT or checkpointed prepared version table, using the Prepared Action Table and the Logged Object List in the checkpoint object. The CPVT maps the virtual address of an object header to the virtual address of its current version, for objects that were written to the log for an action that had prepared or had begun writing object versions to the log as part of prepare at the time of the last checkpoint.

Recovery continues by reading and processing the log backwards from the last outcome record written before the crash until the first outcome record written after the last checkpoint. Then it processes the Prepared Action Table and the Committing Action Table from

the checkpoint object. In the case of a crash during garbage collection, only the tables in the checkpoint object are processed. Each outcome record and entry in a checkpoint table is processed to restore object versions to virtual memory, restore locks, reconstruct information for two-phase commit and compute the highest UID in use based on the information extracted from the previously processed outcome records and the CPVT.

The recovery system organizes the information it needs from the outcome records in three tables. The tables are empty at the beginning of recovery, and are built up incrementally.

1. The PT, or participant action table, maps action identifiers to participant action states. The three possible states are *prepared*, *committed*, and *aborted*. If *prepared*, the entry contains the set of objects modified by the action. The set of modified objects is called the MOS.

2. The CT, or coordinator action table, maps action identifiers to coordinator action states. The two possible states are *committing* and *done*. If *committing*, the entry contains a list of guardian identifiers of the participants.

3. The OT, or object table, maps virtual addresses of resilient object headers to object states. An object state consists of a pair of properties. The first property can take on the values *old* or *new*, depending on whether the object was created before or after the last checkpoint. If the state has the *new* property, the entry contains the new virtual address of the object's header. The second property can take on the values *empty*, *partial* or *restored*, depending on the degree to which the object's value has been restored. The *empty* property indicates that a header has been created for the object, but the object's value has not yet been restored. It is used when a reference to an object created after the last checkpoint is encountered during recovery before the object itself has been restored. *New* is the only value for the first property that is possible with *empty*. The *partial* property indicates that the object's current version has been restored. The *restored* property indicates that the object's base version has been restored. Because the log records are processed in the opposite order in which they were written, the sequence of properties possible for an entry's state is *empty*, *partial*, *restored*[3].

Computing the highest UID in use is simple. Any object that has been made accessible since the last checkpoint will have a base_committed record in the portion of the log processed during recovery. Therefore, it can be calculated by taking the maximum of the highest UID in use at the time of the checkpoint and the UIDs for objects for which a

[3]An implementation could use just two values for the second property – *partial* and *restored*. The third value, *empty*, has been introduced for the sake of clarity.

base_committed record is processed[4].

After the processing of the checkpoint object is complete, the graph of stable objects is traversed starting with the stable root. This traversal serves two purposes: to clean up object headers and locks for built-in atomic objects that were accessed by actions aborted by the crash, and to check that the recovered stable state is consistent. Consistency of the stable state means that all objects encountered during the traversal have valid headers and all pointers encountered during the traversal point to objects that are in a consistent state. Note that, for recovery from a crash during garbage collection, the checks for consistency and the clean up of object headers could take place during the traversal of the stable state required to reconstruct to-space. For the sake of clarity, the algorithm has been described with two independent traversals of the stable state.

After recovery, the information in the PT for actions in the *prepared* state is used by the Argus system to complete two-phase commit for actions for which the guardian was a participant. The information in the CT for actions in the *committing* state is used for actions for which the guardian was the coordinator. The OT is discarded at the end of recovery.

There are several subroutines that the recovery system uses to restore object versions from the log. *Base restore* determines whether the most recent base version for an object has already been restored; if not, it restores the base version. *Prepared restore* restores the current version of an object. Both use *restore version*, which restores a logged version to virtual memory.

All three subroutines need to determine whether an object header is on the backing store given its virtual address. The virtual address of the object's header is compared with the highest virtual address in use at the time of the last checkpoint. If it is less, the header is on the backing store. If it is greater, the object was created and made accessible since the last checkpoint.

Base restore restores the base version for an object if necessary. Its input is a <virtual address of object header, log address> pair. The object is looked up in the OT. There are

[4]The current recovery system assigns UIDs only to resilient objects that are stable. The UIDs are assigned when the object becomes accessible. Assuming the same implementation, the highest UID in use would be the UID for the object recorded in the last base_committed record written to the log before the crash.

three cases:

1. There is no entry in the OT. The virtual address of the object's header is checked to see if the header is on the backing store. If it is, an entry is inserted in the OT in the <*old*, *restored*> state. If not, an empty object header is created for the object and an entry is inserted in the OT in the <*new*, *restored*> state. In both cases, the base version of the object is restored using *restore version*, and a pointer to the base version is placed in the object's header. The locks and the version stack in the object's header are cleared.

2. The object's entry in the OT has the *partial* or *empty* property. The base version of the object is restored using *restore version*, and a pointer to the base version is placed in the object's header. The entry in the OT is changed to have the *restored* property.

3. The object's entry in the OT shows that its state is *restored*. The object's most recent base version before the crash has already been restored. Nothing is done.

Prepared restore is used to restore the current version of an object, for an object that was modified by an action that was prepared at the time of the crash. Its input is a <virtual address of object header, log address> pair. The object is looked up in the OT. There are two cases:

1. There is no entry in the OT. The virtual address of the object's header is checked to see if the header is on the backing store. If it is, an entry is inserted in the OT in the <*old*, *partial*> state. If not, an empty object header is created for the object and an entry is inserted in the OT in the <*new*, *partial*> state. In both cases, the current version of the object is restored using *restore version*, a pointer to the current version is placed in the object's header, and the action is granted a write lock.

2. The object's entry in the OT shows that its state is <*new*, *empty*>. The current version of the object is restored using *restore version*, a pointer to it is placed in the object's header, and the action is granted a write lock. The entry in the OT is changed to have the *partial* property.

Restore version restores an object version to virtual memory if necessary. The input to *restore version* is a <virtual address of object header, log address> pair; its output is the address in virtual memory of the restored version. The log address is the address of the object version to be restored from the log or the log address recorded in the checkpoint object. The latter case indicates that the checkpoint object is being processed; all versions to which the checkpoint object refers are on the backing store. The log address is compared with the log address recorded in the checkpoint object. There are two cases:

1. The log address is less than or equal to the log address recorded in the checkpoint object. The object version to be recovered is intact on the backing store. Its virtual address is retrieved from the CPVT.

2. The log address is greater than the log address recorded in the checkpoint object. The object version has to be restored from log. *Restore version* unflattens and copies the logged version to virtual memory. As it does so, it examines the version for <virtual address of object header, UID> pairs, references to resilient objects copied to the log independently. The virtual address of the object header has to be checked to see if the header is on the backing store. If it is, the address is left unchanged. If not, the OT is searched for an entry for the object. If the object is in the OT, then its entry in the OT has the *new* property and contains the virtual address of its reconstructed header. Otherwise, an empty object header is created for the object and an entry for it is inserted in the object table in the <*new*, *empty*> state. Once all contained references have been processed and restored, *restore version* returns the virtual address of the restored version.

The recovery algorithm for built-in atomic objects can now be described.

1. Consult the guardian manager to find out the location of the backing store for the guardian on secondary storage, the mapping of virtual addresses to backing store blocks and the location of the log on stable storage. Recreate the virtual address space of the guardian. Check if the garbage collection flag is set. If it is, recover the stable state in to-space using the recovery algorithm from the previous chapter. (Recall that the checkpoint object is transported to to-space and its addresses translated as part of that algorithm.) Reset the garbage collection flag.

2. Construct the CPVT from the checkpoint object.

3. Create an empty PT, OT and CT. Make note of the highest virtual address and highest UID in use at the time of the checkpoint from the checkpoint object.

4. Read the log backwards starting with the last outcome record in the log and ending with the first outcome record written after the checkpoint. (This portion of the log is empty if this was a crash in the middle of garbage collection.) Process each outcome record as follows:

   (a) Done record. Extract the action identifier from the record. Insert a *done* entry in the CT for the action identifier appearing in the log record.

   (b) Committing record. Extract the action identifier from the record. Look up the action in the CT. If its state in the CT is *done*, then ignore the record. Otherwise, insert an entry in the CT for the action in the *committing* state with the list of guardian identifiers of participants in the action taken from the committing record.

   (c) Committed record. Extract the action identifier from the record. Insert an entry in the PT for the action in the *committed* state.

(d) Aborted record. Extract the action identifier from the record. Insert an entry in the PT for the action in the *aborted* state.

(e) Prepared record. Extract the action identifier from the record. Look up the action in the PT. There are four cases:

i. The action's state in the PT is *committed.* For each <virtual address of object header, log address of version> pair in the prepared record, do a *base restore.*

ii. The action's state in the PT is *aborted.* Do nothing.

iii. The action is not in the PT. For each <virtual address of object header, log address of version> pair in the prepared record, do a *prepared restore.* Insert an entry in the PT for the action in the *prepared* state with a MOS constructed from the prepared record.

iv. The action's state in the PT is *prepared.* (A prepared-data record for the action has already been processed.) For each <virtual address of object header, log address of version> pair in the prepared record, do *prepared restore.* Add each object listed in the prepared record to the MOS in the PT entry.

(f) Base_committed record. Recompute the highest UID in use by taking the maximum of the UID in this record and the highest UID so far. Do a *base restore* for the <virtual address of object header, log address of this log record> pair. Insert the UID in the reconstructed object's header.

(g) Prepared_data record. Extract the identifier of the action from the record. Look up the action in the PT. There are four cases:

i. The action is in the PT in the *committed* state. Do a *base restore* for the <virtual address of object header, log address of this record> pair.

ii. The action is in the PT in the *aborted* state. Do nothing.

iii. The action is not in the PT. Do a *prepared restore* for the <virtual address of object header, log address of this record> pair. Insert an entry in the PT for the action in the prepared state with this object as the only entry in its list of modified objects.

iv. The action's state in the PT is *prepared.* (A prepared-data record for the action has already been processed.) Do a *prepared restore* for the <virtual address of object header, log address of this record> pair. Add the object to the MOS in the PT entry.

5. Process the checkpoint object.

(a) Look up each action in the Prepared Action Table in the PT. There are four possibilities:

i. The PT shows the action has *committed.* For each object in the object list for the action, do a *base restore* for the <virtual address of object header, log address in checkpoint object> pair.

ii. The PT shows the action has *aborted.* Do nothing.

iii. The action is not in the PT. For each object in the object list associated with the action in the Prepared Action Table, do a *prepared restore* using the <virtual address of object header, log address in checkpoint object> pair. Insert an entry in the PT for the action in the *prepared* state with a MOS constructed from the object list.

iv. The action's state in the PT is *prepared.* (A prepared-data record for the action has already been processed.) For each object in the object list associated with the action in the Prepared Action Table, do a *prepared restore* using the <virtual address of object header, log address in checkpoint object> pair. Add the object to the MOS in the PT entry.

(b) Look up each action in the Committing Action Table in the CT. If the action appears in the CT, then its state is *done* and nothing is done. Otherwise, read the committing record for the action from the log using the pointer to it in the Committing Action Table. Insert the list of guardian identifiers of participants from the committing record in the CT in an entry for the action.

6. Starting with the stable root, traverse the graph of accessible stable objects. As each object is encountered check that it is consistent. A consistent object has a valid descriptor and each of its contained objects is consistent. If an inconsistent object is found, restart recovery using the procedure for a hard crash (explained below.) As each header for a built-in atomic object is encountered, mark it accessible (to recompute the Accessibility Set) and look it up in the OT. There are three possibilities:

   (a) The object's state has the *restored* property. The object and its header have already been restored, do nothing.

   (b) The object's state has the *partial* property. The current version of the object has been restored by recovery together with a lock for the prepared action; do nothing. The last checkpoint guaranteed that the object's base version and the base version pointer in its header are intact.

   (c) The object is not in the OT. Clear the pointers to the current version and the lock in the object header. The last checkpoint guaranteed that the object's base version and the base version pointer in its header are intact.

7. The PT, CT and the highest UID in use are returned to the Argus system.

The procedure for recovery from a hard crash is similar to the recovery algorithm outlined above. The major differences follow. The entire log is read during recovery. An object is identified by its UID rather than the virtual address of its object header. Object versions are always restored from the log. The only purpose served by traversing the stable state at the end of recovery is to recompute the Accessibility Set.

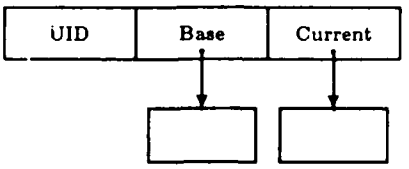


Figure 4.11: Mutex Object

4.8 Recovery of Mutex Objects

A recovery method that does not require the whole log to be scanned after a crash must be devised for mutex objects. A requirement for recovery similar to the one for built-in atomic objects is: a mutex needs to be recovered from the log only if it was written to the log since the last checkpoint. A shadowing representation, similar to the one for built-in atomic objects, whose modification is coordinated with two-phase commit, will allow this requirement to be met.

4.8.1 Representation of Mutex Objects

A mutex object is represented by a header; the header contains the UID of the object, a pointer to the object's base version and a pointer to its current version. Figure 4.11 illustrates the representation. All operations of the mutex type generator modify or observe the current version. The base version exists only for the purpose of recovery. It is a copy of the current version made at the last time the mutex was written to the log by a preparing action, a copy that could be recovered in the event of a crash. The pointer to the new base version is installed at prepare. Other information is required in the header to keep track of mutual exclusion, but it is volatile and its format is not important to recovery.

4.8.2 Writing Mutex Objects to the Log

There is a special outcome record for logging mutexes called the mutex record. Figure 4.12 illustrates it. Its content is similar to that of a base_committed record. The mutex record contains the UID of the mutex, the virtual address of its object header, and the mutex value. The log pointer points to the previous outcome record written to the log. A mutex version is flattened when written to the log with the exception that references to other resilient objects are preserved.

UID
virtual address of object header
mutex value
log pointer

Figure 4.12: Format of Mutex Record

According to the recovery semantics for mutex presented in the chapter on Argus, the value recovered for a mutex must be at least as recent as the value written to stable storage on behalf of the last action that prepared which called *changed* for that mutex. This semantics is achieved by writing a mutex record to the log for an action that has called *changed* after the call, but before the action's prepared record is written to the log. Thus, the value recovered for a mutex can be the value contained in the last mutex record written for it to the log. This behavior is easily achieved using outcome records. All outcome records up to the checkpoint are processed during recovery. Since the log is processed in reverse order, the first outcome record processed for a mutex contains a value that can be recovered.

The procedure for logging a mutex object follows. When an action that has made a mutex accessible from a stable variable or called *changed* prepares:

1. Seize the mutex.
2. Make a copy of the current version of the mutex and install the copy as the base version. The base version is a copy of the current version in which all contained non-resilient objects have been copied. If the base version shared non-resilient objects with other objects in the guardian, the value for the base version could be altered after its installation. In that case, the system could not guarantee that the base version would remain in a consistent state on the backing store after a checkpoint and a subsequent crash.
3. Construct a mutex record containing a flattened copy of the current version and write it to the log.
4. Release the mutex.

Given the shadow representation for a mutex object and the above procedure, a checkpoint guarantees that the base version and the pointer to it in the header survive in a consistent state on the backing store as long a mutex record has not been written to the

log since the checkpoint. Seizing the mutex insures that the version of the mutex written to the log and installed as the new base version is consistent.

The order of the steps taken at prepare is important. The new base version is installed before the record is written to the log to insure consistency between the log and the backing store at a checkpoint. The last base version installed for a particular mutex must always be a version of the mutex that could be recovered if the guardian were to crash. Given the new semantics for mutex, it does not do any harm to create new base versions more frequently than necessary. Note what would happen if the two steps at prepare were reversed and a checkpoint were taken before the base version were installed, but after the record were written to the log. In that case, the mutex record would not be in the portion of the log scanned after a crash and the correct mutex version might not be recovered.

Note that the shadowing representation requires that two versions be in virtual memory for a given mutex at all times. If the compiler and linker for Argus could enforce the restrictions that the contents of a mutex be accessible only while it is seized and that no non-resilient objects contained in a mutex be shared with other objects in the guardian, the period of time during which two versions of the mutex are in virtual memory could be shortened. In that case the current version could be directly installed as the base version during prepare without making a copy. Making the copy could be delayed until the next time the mutex were seized. Other alternatives requiring less storage in virtual memory are discussed in the next chapter.

4.8.3 Recovery of Mutex Objects

The changes required to the recovery algorithm presented for built-in atomic objects to incorporate mutex objects are discussed below.

The OT is extended to contain mutex as well as built-in atomic objects.

There is one more type of outcome record to process. When a mutex record is read from the log, the recovery system calls a subroutine named *mutex restore*. The subroutine decides whether the most recent base version of a mutex object has been restored, if not it restores the base version. Its input is a <virtual address of object header, version log address> pair. It looks up the object in the OT. There are three cases:

1. There is no entry in the OT. The virtual address of the object's header is checked

to see if it is on the backing store. If not, an empty object header is created for the object and an entry is inserted in the OT in the <*new*, *restored*> state and the virtual address of the header is inserted in the entry. If it is, an entry is inserted in the OT in the <*old*, *restored*> state. In both cases, the base version of the object is restored using *restore version*; and a pointer to the base version is installed in the object's header.

2. The object's state has the *empty* property. (A data record referencing the mutex has already been processed and the mutex was created since the last checkpoint.) The base version of the object is restored using *restore version*, and a pointer to the base version is installed in the object's header. The entry in the OT is changed to have the *restored* property.

3. The object's state has the *restored* property. Do nothing. The most recent version for this mutex has already been restored.

When the graph of accessible stable objects is traversed, each mutex object is processed as follows: a copy of the base version in which all contained non-resilient objects are copied is made and installed as the current version. This step is expensive. Its cost could be amortized over the running of the guardian by delaying the copying of the mutex until the first time it is seized after the crash. In that case, a flag would have to be set in the mutex header[5] during the traversal of stable objects warning that a current version has to be installed before the mutex is used.

The last change concerns both the log and the recovery algorithm. Every reference to a resilient object in a data, base_committed, prepared data, or mutex record needs to be tagged with the type of the object being referenced: mutex or built-in atomic. A single bit is enough to hold the tag. Mutex headers are not necessarily the same size as the headers for built-in atomic objects. When a reference to a resilient object that has been created since the checkpoint is found in a logged object version being restored to virtual memory by *restore version*, the recovery system has to know what size header to allocate for it in virtual memory. The alternative is for the recovery system to always allocate enough storage for the larger header.

[5]The pointer to the current version could be set to nil

4.9 Housekeeping the Log

A guardian's log contains a history of the computation at the guardian since it was created. Though the checkpoint mechanism is being used to shorten the time for recovery after a soft crash, it is still desirable to reclaim the storage being used by the log and to keep the time for recovery from a hard crash reasonably short.

A method for shortening the log by creating a new log containing a snapshot of the stable state is presented in a paper on the current recovery system[22]. While the guardian continues normal processing, a snapshot of the stable state is constructed by traversing the stable state in virtual memory, and writing it to a new log. Information written to the old log since the snapshot was initiated is transferred to the new log, translating log addresses as appropriate. When all pertinent information has been transferred to the new log, processing at the guardian is suspended and the old log is replaced by the the new log in an atomic step.

An observation about the structure of the log allows a modification to the procedure presented above in which it is not necessary to install a new log or copy records from an old log to a new log. Remove from consideration all data records that were written to the log by actions that aborted before they finished preparing; these records are never needed. There are prefixes of the resulting log for which all outcome records in the prefix were written to the log on behalf of actions that completed. More specifically, every data record in the prefix has a prepared record that points to it. For every prepared data record in the prefix, there is an associated committed or aborted record. For every committing record in the prefix, there is an associated done record. The observation is that the largest such prefix at the time a snapshot was initiated can be deleted from the log and replaced by a new log segment containing the snapshot.

A new representation is required for the log to make use of the observation. Every log consists of two segments: a prefix and an active portion. When records are written to the log during normal processing, they are always appended to the active portion. The representation invariant is that no outcome record or potential outcome record in the active portion of the log refers to a record in the prefix.

The procedure that makes use of the observation follows. Take a checkpoint. While the

recovery system is suspended during the checkpoint, calculate the log address of the earliest record written to the log for a committing action for which the guardian is the coordinator, or referenced by an action that is prepared or has already written data records to the log as part of prepare. Install the checkpoint. While the guardian continues normal processing, traverse the stable state and write a snapshot of it to a new log prefix on stable storage. Once the snapshot completes, replace the prefix of the log by the new prefix and discard from the active portion of the log all records preceding the log address calculated in the earlier step.

The snapshot can use modified versions of the base_committed and mutex records. No virtual addresses need be recorded in the records; the checkpoint taken at the outset of the snapshot insures that recovery from a soft crash will never need to reconstruct object versions from the snapshot. Also, it is not necessary to chain the records together using log pointers; all records used in the snapshot are outcome records.

For recovery from a soft crash, it is not necessary that the whole log be on-line on fast storage (e.g., disk). Assuming that most crashes will be soft crashes, the snapshot mechanism can be modified to write the new log prefix directly to a less expensive, and less accessible medium such as magnetic tape.

Chapter 5

Alternatives

This chapter discusses several optimizations and alternatives to the recovery scheme presented so far. The first section suggests two ways to decrease the time to take a checkpoint. The second section presents a special case for which the storage required for a mutex can be reduced. The third section outlines an alternative way to locate resilient objects in virtual memory after a crash and compares the alternative to the recovery scheme presented in the last chapter. Finally, the fourth section discusses an alternative for atomic garbage collection that is faster, but requires more storage.

5.1 Cheaper Checkpoints

All processing is halted at a guardian while a checkpoint is taken. It is important that this delay be short, especially the delay for the checkpoint preceding an atomic garbage collection. Almost all of the delay is due to the time required to write the dirty pages of physical memory to the backing store. Two ideas are presented for shortening the delay by decreasing the number of pages that have to be written.

The first idea is to keep track of the pages of virtual memory that have stable objects on them. Define the *Stable Page Set* (SPS) to be the set of virtual memory pages on which there are stable objects. At a checkpoint, only the dirty pages of physical memory that are also in the SPS need to be written to the backing store. There may be a large amount of volatile state in the heap of a guardian and there is no reason to write it to the backing store during a checkpoint.

Maintaining the SPS is similar to maintaining the Accessibility Set. The SPS can contain

a superset of the pages that contain stable objects. The SPS is initialized when a guardian is created. During normal processing, it is updated every time an action prepares. At each prepare, the SPS is augmented by the pages on which newly accessible objects reside, the pages on which current versions for accessible built-in atomic objects that were modified by the action reside and the pages on which the new base versions for accessible mutex objects reside. Membership in the SPS is recalculated at every garbage collection. Recalculating membership for the atomic garbage collection algorithm presented in chapter 3 is easy. When that algorithm completes, the portion of virtual memory in to-space that contains stable objects is contiguous. After a crash, the membership in the SPS can be recalculated during the traversal of the stable state that repairs object headers.

An efficient implementation of the SPS would be to use a bit of the page table entry for each page of virtual memory to indicate membership in the SPS. Then at a checkpoint only those pages whose entries have both the SPS and dirty bits set are written to the backing store.

A second way to shorten the time for a checkpoint is to use the idle time of a guardian to keep the number of dirty pages low. A background process can be set up in the guardian that runs at the lowest priority. The background process continuously scans the guardian's page table writing dirty pages to the backing store.

5.2 Less Storage for Mutexes

There is not much experience with the use of mutexes; but from the known examples[26] it is probable that mutexes will often be large. The scheme presented in chapter 4 doubles the storage for accessible mutexes by requiring that an extra mutex version be kept in virtual memory for the purpose of recovery. This is large space requirement for information that is not required by a normal running guardian. A simple optimization for mutexes that contain no resilient objects is presented in this section. Such mutexes might be used by a program that needs objects to be resilient, but not necessarily atomic. However, there is not yet enough experience writing programs in Argus to determine how frequently mutexes are used in this way.

Normally an object version that was written to the log before the last garbage collection

cannot be recovered from the log after a crash because it could contain references to resilient objects that moved during the garbage collection. However, an object version that does not contain any references to other resilient objects can always be recovered from the log no matter when it was written. This observation suggests a strategy for the recovery of mutexes that contain no resilient objects. For such a mutex, the log address of the mutex's base version is recorded in its header in place of the pointer to its base version in virtual memory. After a crash, these mutexes can be recovered when the stable state is traversed to repair object headers.

The types of the objects that a mutex contains usually will be determined at compile time; at the latest the types are discovered when the guardian is configured. Appropriate code to treat the special case can be linked into the guardian at that time.

5.3 Locating and Identifying Stable Objects

Choosing a method for identifying stable objects and locating them in virtual memory after a crash was a key decision in the design of the recovery system. Using the method described in the previous chapter, a map is written incrementally to the log; all resilient objects are referenced in the log using <UID, virtual address of object header> pairs. An alternative would be to maintain the map of UIDs to virtual addresses in virtual memory. Virtual memory is a reasonable place to keep the map; the map is needed only in the event of a soft crash and virtual memory survives a soft crash. In order for this approach to work, a representation is needed for the map that will survive a crash in a consistent state on the backing store. A similar map called Object Header Storage was used by the Swallow object repository[2]. However the design of the map for Swallow was considerably more complicated than what is presented below because it had to support a wider range of operations.

This alternative scheme for finding resilient objects after a crash was rejected because of the extra cost it imposed for garbage collection. However, its description is included to show the range of options that was explored. The remainder of this section presents the alternative. First the requirements for the object map in virtual memory are presented. An implementation fulfilling those requirements is shown, together with a method for reco-

ering the map after a crash. It is shown how the use of the map can reduce the storage requirements for mutexes. Finally, the object map scheme is compared with the recovery scheme presented in the last chapter.

5.3.1 Keeping a Map in Virtual Memory

A map, used in place of the Object Table during recovery, is stored in virtual memory at all times. It is called the Accessible Object Map or AOM. The AOM maps the UID of a stable object to the virtual address of the object's header and an object state. There is an entry in the map for every resilient object accessible from a stable variable. Like the accessibility set, the AOM might contain entries for objects that are no longer accessible. The map is maintained during the normal running of a guardian, but only used after a crash.

The object state stored in an entry is meaningful only during recovery. It can take on four values *not_recovered*, *empty*, *partial* or *restored*. *Empty*, *partial* and *restored* retain the meaning they had for the OT. *Not_recovered* means that the object has not been recovered since the checkpoint. The *not_recovered* value allows the recovery system to determine which object headers have not yet been restored when it traverses the stable state at the end of recovery. In the first recovery scheme, the OT only had entries for objects restored during recovery; the *not_recovered* value was implied by the absence of an entry in the OT. This does not work for the AOM because the AOM has entries for all accessible objects.

Under the recovery scheme using the AOM, the contents of the log records are changed. No virtual addresses are recorded in the log; resilient objects are identified solely by their UID in the log. When a UID is read from the log during recovery, the AOM is searched for its entry to find the virtual address of its object header.

The operations that the AOM must support are a subset of the operations normally implemented by a map type. The restricted number of operations and the restricted contexts in which they can be called allow a simple implementation of the AOM. The operations needed are *insert*, *lookup*, *modify state* and *rebuild*. The *insert* operation adds a new mapping to the table. It is called by the recovery system during the prepare phase of two-phase commit when a resilient object becomes newly accessible or during recovery when a reference

to a resilient object created since the last checkpoint is processed. The *lookup* operation is called only during recovery to find the location of an object in virtual memory given its UID. The *modify_state* operation is called only during recovery to update the object state associated with an entry. The *rebuild* operation is combined with garbage collection to remove entries from the map for objects that are no longer accessible and to associate a new virtual address with each UID.

5.3.2 Implementation of the AOM

The AOM must be implemented such that it can be recovered in virtual memory after a crash before the log is scanned. It is implemented using a hash table for which space is allocated in the same heap used for other objects.

Figure 5.1 illustrates the representation used for the AOM. The hash table is a sequence of bucket pointers. Each pointer points to the head of the linked list of entries in the bucket. The pointer is nil if the bucket is empty. Each entry in the linked list contains a UID, the associated object state (*not_recovered*, *empty*, *partial*, or *restored*), the virtual address of the object header, and a pointer to the next entry in the list[1]. That pointer is nil if the entry is last in the list. A hash function on the UID determines the bucket in which an entry is inserted. The *lookup* and *modify_state* operation work in the obvious way. The *insert* operation always inserts the new entry at the end of the linked list to which the hash function directs it.

When a checkpoint is taken the AOM must be in a consistent state. Consistency means that the linked list in each bucket is complete. One way to insure consistency is to make sure that a checkpoint is not taken while an insert operation is in progress.

Objects move during garbage collection; hence, the virtual addresses of object headers in the entries in the AOM need to be updated. Copying the AOM to to-space provides an opportunity both to update the entries and to delete entries for objects that are no longer accessible. The AOM is copied using a special algorithm with knowledge of the AOM's representation instead of the usual copying algorithm used for atomic garbage collection.

[1]Since only two bits are needed to encode the object state, an implementation does not need to use a whole memory word to hold the state. Bits from the memory word holding the UID or virtual address can be used instead.

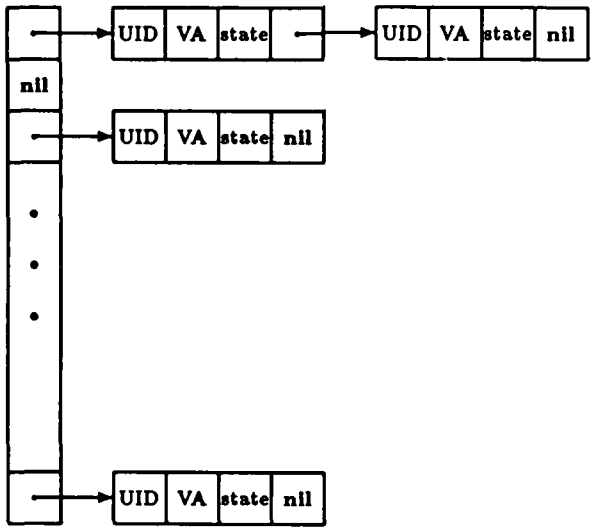


Figure 5.1: Representation of the AOM

The AOM is copied as part of atomic garbage collection after the stable state has been copied, but before the dirty pages of to-space have been written to the backing store.

The algorithm follows. First, space for the sequence of bucket pointers is allocated in to-space equal in size to the sequence in from-space. Then the linked list in each bucket in from-space is traversed. Each entry is processed on the basis of the virtual address of the object header in it. There are two cases:

1. It is the address of the descriptor cell for an object header that contains a forwarding address. The corresponding object is still accessible and the entry is copied to to-space. The forwarding address replaces the virtual address of the object header in the to-space copy.

2. It is the address of the descriptor cell for an object header that contains a valid descriptor. That object is no longer accessible and the entry is not copied.

After the AOM has been rebuilt in to-space, atomic garbage collection continues in the usual way by writing all dirty pages of to-space to the backing store. This insures that the rebuilt AOM is in a consistent state on the backing store for to-space.

This algorithm rebuilds the AOM in a way that improves locality of reference for subsequent operations. Entries in one linked list are copied to the same page of virtual memory with high probability.

5.3.3 Recovering the AOM

During recovery, the AOM must be recovered to a consistent state before processing of the log can begin. It can be returned to the state it was in at the time of the last checkpoint. Any insertion since the last checkpoint was for an object that became newly accessible since the last checkpoint. Such an object must have a base_committed or mutex record in the portion of the log written since the last checkpoint. Its entry can be reinserted during the processing of the log.

Recall that the highest virtual address in use at the time of the last checkpoint is recorded in the checkpoint object. Any entry in the AOM that was inserted in the AOM since the last checkpoint will be at a higher virtual address. To return the AOM to its state at the time of the checkpoint, the linked list in each bucket is traversed. If the pointer to the next entry in a list has an address greater than the highest virtual address at the time of the checkpoint, that pointer is replaced by nil to cut off the part of the list inserted since the checkpoint. As each entry is traversed, the object state associated with it is set to *not_recovered.*

The AOM will be recovered correctly even if a crash occurs in the middle of recovery. New entries are always inserted at the end of the linked lists; thus, the initial portion of the linked lists at the time of the last checkpoint survives a crash. Object states do not have to survive a crash; they are always reset to *not_recovered* at the beginning of recovery.

After the AOM has been recovered, the log can be processed. During the processing of the log, the AOM is used in place of the OT and to find resilient objects in virtual memory.

5.3.4 Recovery of Mutexes

The AOM allows more flexibility in the choice of recovery schemes for mutex objects. The recovery scheme presented for mutex in the last chapter required that two versions of the mutex be kept in virtual memory. Using the AOM, only a current version and enough information to recover a base version need to be kept in virtual memory.

The new representation used for mutex objects is similar to the representation presented in the previous chapter. However, instead of a base version, information about the last version written to the log is kept. That information includes the log address of that version

and a list of the virtual addresses of the object headers for the resilient objects contained in that version. The log address allows mutex versions to be recovered without having to scan the whole log [2]. The list of contained objects is necessary because of garbage collection. It prevents resilient objects that are accessible from the logged version, but no other object in virtual memory, from being treated as garbage and removed from virtual memory and the AOM.

To ensure consistency between virtual memory and the log, a mutex header cannot be updated with a new log address and a list of contained objects until the corresponding mutex record is in the log. Thus, logging an accessible mutex object and updating its header is a two step process during prepare for an action that called *changed*.

In the first step, the mutex is logged. While the mutex is seized,

- Construct a list of resilient objects contained in the current version. Remember the pointer to the list.
- Construct a mutex record from its current version and write it to the log. Remember the log address of the record.

After the prepared record for the action has been forced to the log, the mutex record is physically in the log and the mutex header can be updated in the second step.

- Compare the remembered log address with the log address in the header. If it is greater, install it and the remembered pointer to the list of contained objects in the mutex header. This comparison is necessary because two actions could write mutex records for the same mutex in one order, but write their prepared records in the opposite order.

The log and the backing store must be kept consistent at a checkpoint. At a checkpoint, the second step outlined above is performed for any mutex for which the first step has been carried out. This takes place after the log has been forced, but before the checkpoint object is installed. Forcing the log during a checkpoint ensures that the mutex record is in the log.

After a crash, the mutexes for which a mutex record was written to the log since the last checkpoint are recovered while the log is scanned. All three components of the header are recovered – log address, list of contained objects and current version. There are two choices for recovering the remaining mutexes. They can be recovered when the stable state

[2]The inclusion of the log address in the header also allows the original Argus recovery semantics for mutex to be implemented. However, recovery is described using the semantics assumed for this thesis.

is traversed to repair object headers or they can be recovered the first time that they are seized after the crash. In both cases the log address in mutex header is used to find them in the log. Note, that the checkpoint guarantees that these log addresses are on the backing store.

5.3.5 Comparison With First Solution

The main advantages of the scheme using the AOM with respect to the scheme presented in the previous chapter are that less storage is required for mutexes and that two-phase commit is faster because less information needs to be written to the log. The main disadvantage is the extra expense incurred during garbage collection to rebuild the AOM. The difference in storage requirements is not large, as will be shown below. More effective schemes for saving storage devoted to mutex versions are presented in chapter 6. It is hard to compare the overhead for garbage collection and two-phase commit, but it is likely that the extra overhead for garbage collection is the greater expense.

Using the first scheme for recovery, two mutex versions must be kept in virtual memory for each mutex accessible from the stable variables. The scheme using the AOM only needs to maintain one full mutex version in virtual memory plus a list of resilient objects accessible from the last logged version. However, this is not always as big a savings as one might hope.

Mutex objects are usually used to implement user defined types. In most cases when a mutex is used to implement a user defined atomic type, it will contain as its largest component a mutable collection of built-in atomic objects. The built-in atomic objects are required to allow the implementation to find out about the commits and aborts of the actions that use it. A typical example is the implementation of the semi-queue type [26]: each item on the queue is kept in an atomic variant to keep track of the status of the action that enqueued or dequeued it. The storage required to hold a mutable collection of built-in atomic objects is comparable to the storage required to hold a list of [illegible] objects.

The savings in storage for mutexes under the AOM scheme [illegible] that contain no resilient objects. Using the [illegible] scheme for recovery and the AOM scheme [illegible]

that contain no resilient objects.

Using the AOM scheme, log records will be shorter because virtual addresses do not have to be recorded in them. The biggest savings is for a data record for a resilient object containing a collection of other resilient objects; it would be close to half the size. The savings for a prepared record for an action that modified two objects would be about twenty percent. The size of aborted, committed, committing and done records would not change. Shorter records mean less time spent writing records to the log and possibly a lower latency for two-phase commit. For an Argus implementation to be successful it is important that two-phase commit be fast.

However, the time savings for shorter records is probably not significant. Note that the time to compose log records will be about the same; when constructing a data record, the dominant cost in construction is reading the headers of contained objects in virtual memory to find out whether they are resilient. Also, the time saved actually writing to the log is small. When writing to a disk, the rotational latency and seek time are very large compared to the time to transfer the data. Because the log is sequential, in both the first scheme and the AOM scheme the same delays due to latency and seek times will be incurred.

Garbage collection is more expensive using the AOM. All of the entries in the AOM have to be updated at every garbage collection whether or not the corresponding objects have been modified since the garbage collection. The first scheme amortizes the updating of the map over the running of the guardian. It writes <UID, virtual address of object header> pairs to the log as log records are written. Only pairs for resilient objects that have been modified or referenced by objects that have been modified need to be written to the log.

[illegible] at the guardian is stopped during garbage collection. Thus, an [illegible] decreases total processing throughput at a guardian. [illegible] be fast. For this reason, it is preferable to [illegible] during garbage collection. If an incremental [illegible] might change

5.4 Garbage Collection

The atomic garbage collector presented in chapter 3 traded time for space. It increased the garbage collection time for stable objects to avoid allocating an extra cell in all objects to hold a forwarding pointer. This section presents a scheme for garbage collection that increases storage requirements, but runs faster. This second scheme might be preferable to the first scheme. However, the first scheme might be more easily adapted to garbage collection for large address spaces. Recovery for large address spaces is discussed in chapter 6.

In this second scheme, any garbage collection algorithm can be used. Before garbage collection commences, a new backing store is acquired on disk. During garbage collection, the first time a page fault occurs for a page of virtual memory, the page is read from the old backing store. However, when dirty pages of main memory are paged out, they are written to the new backing store. When garbage collection completes, a checkpoint is taken using the new backing store and the new backing store replaces the old backing store. If a crash occurs before garbage collection completes, the new backing store is discarded and recovery proceeds using the old backing store.

It is not clear which way of doing atomic garbage collection is best. The original method increased the time overhead for garbage collection, but only for the stable objects. The new method requires double the disk storage while garbage collection is in progress. This increase in storage depends on the size of the heap and not the number of stable objects. However, the increased need for disk storage is only temporary and not all guardians at a node need to garbage collect at once. If guardians do not garbage collect often, the extra disk space can be amortized over all the guardians at a single node.

Chapter 6

Conclusions

This thesis has presented a new, faster recovery method for Argus. The new method is applicable to other systems similar to Argus: systems that use actions to manipulate resilient data in a garbage-collected heap in virtual memory.

Virtual memory consists of a volatile component in main memory and a non-volatile component on disk used as a backing store. The new recovery method distinguishes between crashes in which both the main memory and the backing store are corrupted, and crashes in which only the main memory is corrupted. The former are called hard crashes, the latter soft crashes. Soft crashes are much more frequent than hard crashes. The new method provides fast recovery for soft crashes.

Recovery reconstructs the stable state of an Argus guardian in virtual memory by using the surviving backing store and reading just enough of a log on stable storage to recover information lost from the volatile main memory. Checkpoints, during which all the dirty pages of main memory are written to the backing store, reduce the amount of log that has to be read after a crash. Checkpoints are cheap and several optimizations for making them even cheaper have been suggested.

The log organization used for stable storage is based on the log used by the current recovery system[21,22]. However, an improved method for reclaiming stable storage used by the log has been presented. That method is based on the idea of using two log segments on stable storage to implement a log. The first log segment contains the last snapshot of the stable state. The second log segment contains records written to the log since the last snapshot commenced.

Garbage collection presents two problems for recovery using virtual memory. Because objects move during garbage collection when the heap in virtual memory is compacted, the first problem is locating objects after a crash. Enough information must be available after a crash to find objects on the surviving backing store. Two solutions to this problem have been presented. The preferred solution writes an incremental object map to the log. In that solution, every reference to a resilient object in the log contains the virtual address of the object's header. An alternative solution maintains an object map in virtual memory.

The second problem is the possibility of a crash during garbage collection. Two solutions were also presented for this problem. The first is an atomic garbage collection algorithm based on copying garbage collection. The second solution is an atomic garbage collector that uses a new backing store during garbage collection. The second solution is faster, but it requires more disk storage.

The remainder of this chapter is devoted to a discussion of changes to Argus assumed by this thesis, changes that would reduce the storage requirements for mutexes, a comparison of recovery using virtual memory with the current recovery system, and a presentation of ideas for future work.

6.1 Changes to Argus

This thesis assumed two changes to the Argus programming language. The first change restricts the types of the stable variables. The second change simplifies the recovery semantics for mutexes. This section discusses these changes and two other changes that could reduce the storage required for mutex versions.

6.1.1 Resilient Objects

This thesis restricts the type of a stable variable to a type that can be *guaranteed* to be resilient by static type checking. The restriction is necessary because non-resilient objects are not recovered after a crash. If a non-resilient object became accessible and were subsequently modified, recovery using virtual memory could not ensure the consistency of the object in the recovered heap. Furthermore, the object could contain a pointer to another object that would no longer exist after recovery. Thus, the recovered heap could be broken.

The Argus reference manual proposes a guideline for programs similar to this restriction[17], but the guideline is less restrictive because it does not have to be enforced. The manual's guideline allows the stable variables to be of any type that *could be* resilient. One kind of resilient object allowed by the manual's guideline is a mutable object that is not modified after it becomes accessible from the stable variables. This includes normally mutable and non-resilient objects such as arrays and records. Recovery using virtual memory could allow these objects to be accessible only if it were possible to guarantee that they would not be modified after becoming accessible. Run time enforcement of the guarantee is too expensive.

Note that the restriction assumed in this thesis does not deny a programmer computational power; a non-resilient object can be made resilient by surrounding it with a mutex. The issue is efficiency. A program uses a mutable representation for an immutable type for efficiency reasons. It is convenient to use mutation to create an object of the type even though the object will not be mutated by subsequent operations. In the short term, the best way to circumvent the restriction is to surround these kinds of mutable objects by a mutex or to copy them to immutable objects before they are made accessible. A long term solution might be to investigate methods for a compiler to detect that an immutable type is being implemented using a mutable representation.

6.1.2 Mutexes

The all or none recovery property for mutexes requires that if *changed* is called by an action for several mutexes at a single guardian, either all of the mutex versions written to stable storage on behalf of that action will be recovered after a crash or none of them will. In practice it is difficult to make use of the property and it has not yet been used in Argus programs. Dropping this property from the language simplifies recovery for mutexes. Dropping the property also allows two optimizations that decrease the storage used for extra mutex versions, provided additional restrictions are made:

1. If it can be guaranteed that the contents of a mutex is accessed only while it is seized, and that no non-resilient object contained in it is shared, then the extra version is needed in virtual memory only if the mutex has been seized since the last time that an action that called *changed* prepared. The restriction required for this optimization is one that is already recommended by the Argus reference manual[17] as a proper

programming practice. Mutual exclusion would be violated if the contents of a mutex were accessible outside of a seize, or if objects accessible from a mutex were shared with other objects.

2. It is usually the case that an object's representation is read more often than it is modified. Suppose that in addition to the restriction assumed above, it were possible to differentiate between seizing a mutex for reading and seizing a mutex for writing. Then the extra version is needed in virtual memory only if the mutex has been seized for writing since the last time that an action that called *changed* prepared.

Enforcing the restrictions necessary for the optimizations might require static and dynamic checking and/or changes to the Argus programming language. This is a topic for further research.

6.2 Comparison With Current Recovery System

In comparing recovery using virtual memory with recovery using the log (the current Argus recovery system) there are three factors that need to be considered: the speed of recovery, the resources required by the recovery system during normal execution, and the robustness of the recovery system.

Recovery is much faster using virtual memory than it is using only the log. The work to be done in the case of recovery using virtual memory consists of two parts:

1. Process the part of the log written since the last checkpoint.
2. Traverse the stable state to restore object headers.

The work to be done in the case of the recovery using the log also consists of two parts:

1. Process the part of the log written since the last snapshot.
2. Process the snapshot in the log.

In both cases the time for the second step depends on the size of the stable state. Also the time required for second step is usually small compared to the time required for the first step. Thus, the relative speed of recovery for the two schemes depends on the relative frequency of snapshots in the current system versus checkpoints in recovery using virtual memory.

Checkpoints can be taken much more frequently than snapshots; therefore recovery using virtual memory is a lot faster than recovery using the log. Checkpoints can be taken

more frequently than snapshots because they are cheaper. A checkpoint only requires that dirty pages of physical memory that contain stable objects be written to the backing store, whereas a snapshot requires that the stable state be traversed and written to the log. Note that traversing the stable state also requires bookkeeping to mark objects that have already been copied. Thus, snapshots require modification both of the log and the recovery system.

Faster recovery is not free. Recovery using virtual memory requires more time when a guardian is up and running than does the current recovery system. It requires more space in virtual memory for mutex versions and garbage collection is more expensive.

The extra storage cost depends on the size of the stable state. Extra mutex versions are required only for accessible mutexes. Several schemes for the reduction of the storage requirements for mutexes were presented in the thesis. These schemes depend on restricting the use of mutexes in programs. If practical methods for enforcing these restrictions can be found, the extra storage needed for mutexes will be small.

The extra costs for garbage collection depend on the algorithm used. Three methods for garbage collection were presented: turning a crash during garbage collection into a hard crash, atomic garbage collection using a new backing store, and atomic garbage collection based on copying.

The first two methods take the same amount of time. Both use any garbage collection algorithm followed by a checkpoint. Assuming that heaps are small enough to use "stop the world" garbage collection and that the proportion of time spent garbage collecting is small, atomic garbage collection using a new backing store is preferable to turning a crash during garbage collection into a hard crash. The extra disk storage required during garbage collecting is a small price to pay for fast recovery.

The choice between the last two methods depends on the amount of main memory available to a guardian. Atomic garbage collection using a new backing store is best when most of a guardian's heap fits into main memory. The only cost it adds over normal garbage collection is the cost of the checkpoint. Atomic garbage collection based on copying garbage collection saves space but increases the time. The extra time is proportional to the size of the stable state. This method is preferable for a guardian whose heap does not fit into main memory. In that case the extra I/O time required to page the heap will dwarf the extra

processor time required by the algorithm. A good implementation [illegible] overlap the extra processing time with the I/O as much as possible.

Recovery using virtual memory is slightly less robust than recovery using [illegible] the contents of virtual memory is manipulated directly by machine instructions. This is the assumption that a crash occurs before bad information is written to the backing store might not always be valid. Many crashes are caused by failures of system software rather than hardware. When system software fails, it is possible that page tables for guardians could be corrupted and cause the corruption of a guardian's backing store before the system itself crashed. A guardian's backing store could become corrupted in this manner without being detected by the hardware checks performed by a disk.

Note, however, that it is highly unlikely that the backing store would be corrupted in a way that preserves the consistency of the stable object graph. The algorithm outlined in this thesis checks the consistency of the recovered object graph at the end of recovery when it traverses the stable state to repair object headers. If an inconsistency is found during the traversal, recovery using virtual memory is aborted. In that case, the crash is treated as a hard crash and recovery is restarted using the log.

6.3 Future Work

Implementing the recovery system outlined in this thesis would require extensive changes to UNIX[1] or the implementation of an operating system kernel tailored to Argus. The design of an Argus kernel has already been considered for other reasons, primarily because the overhead for inter-guardian communication in UNIX is too high. A preliminary design for an Argus kernel for the VAX[2] has been described, but not implemented[1]. The design has to be reworked to incorporate changes required to recover using virtual memory.

A major component of the added cost for garbage collection is the associated checkpoint. Although checkpoints are inexpensive when compared with snapshots, they are not cheap when compared with time for garbage collection and when the frequency of garbage collection is taken into account. A busy guardian can be expected to garbage collect several

[1]UNIX is a trademark of Bell Laboratories
[2]VAX is a trademark of Digital Equipment Corporation

times a minute. The frequency of checkpoints could be reduced by decoupling checkpoints from garbage collection. One approach would be to divide the heap in two: a stable part and a volatile part. The volatile part could be garbage-collected frequently using normal methods. Atomic garbage collection could be used less often and only to reclaim storage from the stable part.

Recovery using virtual memory could be extended to work for large virtual address spaces. When recovering a large address space, traversing the whole stable state to restore object headers is too slow. By associating crash counts with object headers, object headers can be restored after a crash as they are accessed during normal processing. Garbage collection is the major problem for large address spaces; an incremental garbage collector that is atomic would have to be developed. It would have to be incremental because the delays experienced during "stop the world" garbage collection would no longer be acceptable. It would have to be atomic since incremental garbage collection would constantly be in progress.

The popular approach to incremental garbage collection is to divide the address space into regions that are collected independently using copying garbage collection[3,20,15]. An atomic incremental garbage collector could use an algorithm for copying similar to the one developed in this thesis. Objects are allocated to the regions according to their age. Researchers have found that younger (recently allocated) objects are more likely to be collected as garbage than older objects and have concluded that "young" regions should be collected more often than "old" regions[20,15,25]. In addition to segregating object by age, an atomic garbage collector might designate some regions to be stable and some volatile as discussed above.

Keeping track of inter-region references efficiently is hard when the heap is divided into regions. The Lisp Machine uses special hardware for this purpose[20]. An atomic garbage collector would have to use data structures that survive soft crashes to keep track of inter-region references between stable regions.

Built-in atomic objects are well suited for recovery using virtual memory. The use of versions in their representation is natural both for them and recovery. Mutex objects are less suited for recovery. The base versions for mutex objects find their use only in recovery.

Several changes or restrictions on the use of mutex objects in Argus have been presented that reduce the storage requirements for mutexes. A way to enforce these restrictions needs to be found. Other researchers have found deficiencies with mutex for designing user defined atomic types[11]. Research into new constructs for building user defined atomic types should also take recovery into account.

Bibliography

[1] L. W. Allen. *Design of a Kernel for Argus.* Programming Methodology Group Memo 43, Laboratory for Computer Science, MIT, Cambridge, Ma., June 1985.

[2] G. C. Arens. *Recovery of the Swallow Repository.* Technical Report MIT/LCS/TR-252, Laboratory for Computer Science, MIT, Cambridge, Ma., January 1981.

[3] H. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[4] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[5] J. Cohen. Garbage Collection of Linked Data Structures. *Computing Surveys*, 13(3):341–367, September 1981.

[6] J. L. Dawson. Improved Effectiveness from a Real Time Lisp Garbage Collector. In *Proceedings 1982 ACM Symposium on Lisp and Functional Programming*, pages 159–167, 1982.

[7] J. L. Eppinger and A. Z. Spector. *Virtual Memory Management for Recoverable Objects in the TABS Prototype.* Technical Report CMU-CS-85-163, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., December 1985.

[8] R. R. Fenichel and J. C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[9] J. N. Gray. *Notes on Database Operating Systems*, pages 393–481. Volume 60 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1978.

[10] J. N. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[11] I. Grief, R. Seliger, and W. Weihl. *A Case Study of CES: A Distributed Collaborative Editing System Implemented in Argus.* Programming Methodology Group Memo 55, Laboratory for Computer Science, MIT, Cambridge, Ma., April 1987.

[12] D. E. Knuth. *Fundamental Algorithms.* Volume I of *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., 1973.

[13] B. W. Lampson. *Atomic Transactions*, pages 246–265. Volume 105 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1981. This is a revised version of Lampson and Sturgis's unpublished *Crash Recovery in a Distributed Data Storage System*.

[14] B. W. Lampson and H. E. Sturgis. Crash Recovery in a Distributed Data Storage System. 1976. version of paper that was not published.

[15] H. Lieberman and C. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[16] B. Liskov. *Overview of the Argus Language and System*. Programming Methodology Group Memo 40, Laboratory for Computer Science, MIT Cambridge, Ma., February 1984.

[17] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl. *Argus Reference Manual*. Programming Methodology Group Memo 54, Laboratory for Computer Science, MIT, Cambridge, Ma., March 1987.

[18] B. Liskov, P. Johnson, and R. Scheifler. Implementation of Argus. 1987. in preparation.

[19] M. L. Minsky. *A LISP Garbage Collector Algorithm Using Serial Secondary Storage*. AI Memo 58, MIT AI Lab., October 1963.

[20] D. Moon. Garbage Collection in a Large Lisp System. In *Proc. of the 1984 Symposium on Lisp and Functional Programming*, pages 235–246, 1984.

[21] B. Oki. *Reliable Object Storage to Support Atomic Actions*. Technical Report MIT/LCS/TR-308, Laboratory for Computer Science, MIT, Cambridge, Ma., May 1983.

[22] B. Oki, B. Liskov, and R. Scheifler. Reliable Object Storage to Support Atomic Actions. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 147–159, December 1985.

[23] P. M. Schwarz. *Transactions on Typed Objects*. PhD thesis, Carnegie-Mellon University, December 1985. Available as Technical Report CMU-CS-84-166.

[24] A. Z. Spector, D. Daniels, D. Duchamp, J. L. Eppinger, and R. Pausch. Distributed Transactions for Reliable Systems. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 127–146, December 1985.

[25] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.

[26] W. Weihl and B. Liskov. Implementation of Resilient, Atomic Data Types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

OFFICIAL DISTRIBUTION LIST

Director
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

Director, Code 2627
Naval Research Laboratory
Washington, DC 20375

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314

National Science Foundation
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

Dr. E.B. Royce, Code 38
Head, Research Department
Naval Weapons Center
China Lake, CA 93555

Dr. G. Hopper, USNR
NAVDAC-OOH
Department of the Navy
Washington, DC 20374

END

FEB.

1988

DTIC